ASN1C

ASN.1 Compiler
Version 5.7
C/C++ Compiler
User's Manual

Objective Systems, Inc. - August 2004

The software described in this document is furnished under a license agreement and may be used only in accordance with the terms of this agreement.

Copyright Notice

Copyright © 1997-2005 Objective Systems, Inc.

All Rights Reserved

This document may be distributed in any form, electronic or otherwise, provided that it is distributed in its entirety and that the copyright and this notice are included.

Author's Contact Information:

Comments, suggestions, and inquiries regarding ASN1C may be submitted via electronic mail to info@obj-sys.com.

CHANGE HISTORY

Date	Author	Version	Description
8/12/2004	ED	5.7	Initial version
1/17/2005	ED	5.72	Updated PER C encode/decode documentation to not use references to deprecated function.
5/27/2005	ED	5.74	Update Extension Elements text in SEQUENCE section to refer to *extElem1* type as *Asn1RTDList* instead of *ASN1OpenType.*
6/07/2005	AB	5.74	Added *"Encoding a Series of Messages Using the C Encode Functions"* section in "Generated BER Encode Functions" chapter.
6/07/2005	AB	5.74	Added *"Decoding a Series of Messages Using the C Decode Functions"* section in "Generated BER Decode Functions" chapter..
6/07/2005	AB	5.74	Updated "*Decoding a Series of Messages Using the C++ Control Class Interface*" section in "Generated BER Decode Functions" chapter.

TABLE OF CONTENTS

Overview of ASN1C 1

Using the Compiler 3

RUNNING ASN1C FROM THE COMMAND-LINE 3
USING THE GUI WIZARD TO RUN ASN1C 11
COMPILING AND LINKING GENERATED CODE 21
PORTING RUN-TIME CODE TO OTHER PLATFORMS 22
COMPILER CONFIGURATION FILE 23
COMPILER ERROR REPORTING 29

Generated C/C++ Source Code 31

HEADER (.H) FILE 31
BOOLEAN 35
INTEGER 35
BIT STRING 37
OCTET STRING 40
ENUMERATED 42
NULL 43
OBJECT IDENTIFIER 44
RELATIVE-OID 45
REAL 45
SEQUENCE 45
SET 51
SEQUENCE OF 52
SET OF 55
CHOICE 55
Open Type 58
Character String Types 59
Time String Types 60
EXTERNAL Type 61
EMBEDDED PDV Type 61
Parameterized Types 62
Value Specifications 64
CLASS specification 69
ABSTRACT-SYNTAX and TYPE-IDENTIFIER 74
Information Object 75
Information Object Set 77
Encode/Decode Function Prototypes 79
Generated C++ Control Class Definition 80
Generated Methods 82
Generated Information Object Table Structures 82
GENERATED C SOURCE FILES 94
Maximum Lines per File 94
Use of the -maxcfiles Option 94
GENERATED C++ FILES 95
GENERATED C/C++ FILES AND THE –COMPAT OPTION 96

GENERAL PROCEDURES FOR ENCODING AND DECODING 98
Dynamic Memory Management *98*
Populating Generated Structure Variables for Encoding *102*
Accessing Encoded Message Components 103
GENERATED BER ENCODE FUNCTIONS 105
Generated C Function Format and Calling Parameters *105*
Generated C++ Encode Method Format and Calling Parameters *106*
Procedure for Calling C Encode Functions *106*
Encoding a Series of Messages Using the C Encode Functions *109*
Procedure for Using the C++ Control Class Encode Method *110*
Encoding a Series of Messages Using the C++ Control Class Interface *112*
GENERATED BER DECODE FUNCTIONS 114
Generated C Function Format and Calling Parameters *114*
Generated C++ Decode Method Format and Calling Parameters *114*
Procedure for Calling C Decode Functions *115*
Decoding a Series of Messages Using the C Decode Functions *117*
Procedure for Using the C++ Control Class Decode Method *119*
Decoding a Series of Messages Using the C++ Control Class Interface *120*
BER Decode Performance Enhancement Techniques *123*
BER/DER Deferred Decoding *125*
GENERATED BER STREAM-ORIENTED ENCODE FUNCTIONS 127
Generated Stream-Oriented C Function Format and Calling Parameters *127*
Procedure for Calling Stream-Oriented C Encode Functions *128*
Encoding a Series of Messages Using the Stream-Oriented C Encode Functions *129*
Generated Stream-Oriented C++ Encode Method Format and Calling Parameters *131*
Procedure for Using the Stream-Oriented C++ Control Class Encode Method *131*
Encoding a Series of Messages Using the Stream-Oriented C++ Control Class Interface *132*
GENERATED BER STREAM-ORIENTED DECODE FUNCTIONS 134
Generated Stream-Oriented C Function Format and Calling Parameters *134*
Procedure for Calling Stream-Oriented C Decode Functions *135*
Decoding a Series of Messages Using the Stream-Oriented C Decode Functions *137*
Generated Stream-Oriented C++ Decode Method Format and Calling Parameters *139*
Procedure for Using the Stream-Oriented C++ Control Class Decode Method *140*
Decoding a Series of Messages Using the C++ Control Class Interface *142*
GENERATED PER ENCODE FUNCTIONS 144
Generated C Function Format and Calling Parameters *144*
Generated C++ Encode Method Format and Calling Parameters *144*
Populating Generated Structure Variables for Encoding *145*
Procedure for Calling C Encode Functions 145
Procedure for Using the C++ Control Class Encode Method 147
Encoding a Series of PER Messages using the C++ Interface *151*
GENERATED PER DECODE FUNCTIONS 151
Generated C Function Format and Calling Parameters *151*
Generated C++ Decode Method Format and Calling Parameters *152*
Procedure for Calling C Decode Functions *152*
Procedure for Using the C++ Control Class Encode Method 154
Decoding a Series of Messages Using the C++ Control Class Interface 156
Performance Considerations: Dynamic Memory Management 157

GENERATED XER ENCODE FUNCTIONS . . . 158
Generated C Function Format and Calling Parameters . . . 158
Generated C++ Encode Method Format and Calling Parameters . . . 159
Populating Generated Structure Variables for Encoding . . . 159
Procedure for Calling C Encode Functions . . . 159
Procedure for Using the C++ Control Class Encode Method . . . 161
GENERATED XER DECODE FUNCTIONS . . . 163
Procedure for Using the C Interface . . . 165
Generated C Function Format and Calling Parameters . . . 165
Procedure for Calling C Decode Functions . . . 166
Procedure for Using the C++ Interface . . . 168
Procedure for Interfacing with Other C and C++ XML Parser Libraries . . . 169
GENERATED XML ENCODE AND DECODE FUNCTIONS . . . 170
GENERATED PRINT FUNCTIONS . . . 173
Print to Standard Output . . . 173
Print to String . . . 173
Print to Stream . . . 174
GENERATED COMPARE FUNCTIONS . . . 176
GENERATED COPY FUNCTIONS . . . 177
GENERATED TEST FUNCTIONS . . . 181
GENERATED MAKEFILE . . . 183
EVENT HANDLER INTERFACE . . . 185
How it Works . . . 185
How to Use It . . . 186
IMPORT/EXPORT OF TYPES . . . 192
ASN.1 to XML Schema Conversion . . . 193
MAPPING OF TOP-LEVEL CONSTRUCTS . . . 193
MAPPING OF ASN.1 TYPES . . . 193
BOOLEAN . . . 193
INTEGER . . . 194
BIT STRING . . . 195
OCTET STRING . . . 197
Character String Types . . . 198
Time String Types . . . 199
ENUMERATED . . . 200
NULL . . . 201
OBJECT IDENTIFIER . . . 201
RELATIVE-OID . . . 202
REAL . . . 202
SEQUENCE . . . 203
SET . . . 206
SEQUENCE OF / SET OF . . . 206
CHOICE . . . 209
Open Type . . . 210
Tagged Type . . . 211
EXTERNAL and EmbeddedPDV Type . . . 212
ASN1C90 . . . 213

ROSE OPERATION and ERROR *213*
SNMP OBJECT-TYPE *215*
Index 217

Overview of ASN1C

The ASN1C code generation tool translates an Abstract Syntax Notation 1 (ASN.1) source file into computer language source files that allow ASN.1 data to be encoded/decoded. This release of the compiler includes options to generate code in four different languages: C, C++, C#, or Java. This manual discusses the C and C++ code generation capabilities. The *ASN1C Java User's Manual* discusses the Java code generation capability. The *ASN1C C# User's Manual* discusses the C# code generation capability.

Each ASN.1 module that is encountered in an ASN.1 source file results in the generation of the following two types of C/C++ language files:

1. An include (.h) file containing C/C++ typedefs and classes that represent each of the ASN.1 productions listed in the ASN.1 source file, and

2. A set of C/C++ source (.c or .cpp) files containing C/C++ encode and decode functions. One encode and decode function is generated for each ASN.1 production. The number of files generated can be controlled through command-line options.

These files, when compiled and linked with the ASN.1 low-level encode/decode function library, provide a complete package for working with ASN.1 encoded data.

ASN1C works with the version of ASN.1 specified in ITU-T international standards X.680 through X.683. It generates code for encoding/decoding data in accordance with the following encoding rules:

- Basic Encoding Rules (BER), Distinguished Encoding Rules (DER), or Canonical Encoding Rules (CER) as published in the ITU-T X.690 standard.

- Packed Encoding Rules (PER) as published in the ITU-T X.691 standard. Both aligned and unaligned variants are supported via a switch that is set at run-time.

- XML Encoding Rules (XER) as published in the ITU-T X.693 standard.

Additional support for XML is provided in the form of an option to generate an equivalent XML Schema Definitions (XSD) file for a given ASN.1 specification. Encoders and decoders can then be generated using the -xml option to format or parse XML documents that conform to this schema. This level of support is closer to the W3C definition of XML then is the ITU-T X.693 XER definition.

The compiler is capable of parsing all ASN.1 syntax as defined in the standards. It is capable of parsing advanced syntax including Information Object Specifications as defined in the ITU-T X.681 standard as well as Parameterized Types as defined in ITU-T X.683. This version of the compiler is also capable of using table constraints as define it ITU-T X.682 to generate single-step encoders and decoders that can encode or decode multi-part messages in a single function call.

This release of the compiler contains a special command-line option - -1990 - that allows compilation of deprecated features from the older X.208 and X.209 standards. These include the ANY data type and unnamed fields in SEQUENCE, SET, and CHOICE types. This version can also parse type syntax from common macro definitions such as the OPERATION and ERROR macros in ROSE.

Using the Compiler

Running ASN1C from the Command-line

The ASN1C compiler distribution contains command-line compiler executables as well as a graphical user interface (GUI) wizard (Windows version only) that can aid in the specification of compiler options. This section describes how to run the command-line version; the next section describes the GUI.

To test if the compiler was successfully installed, enter `asn1c` with no parameters as follows (note: if you have not updated your PATH variable, you will need to enter the full pathname):

```
asn1c
```

You should observe the following display (or something similar):

```
ASN1C Compiler, Version 5.7x
Copyright (c) 1997-2004 Objective Systems, Inc. All Rights Reserved.

Usage: asn1c <filename> <options>

    <filename>          ASN.1 source file name(s).  Multiple filenames
                          may be specified.  * and ? wildcards are allowed.

  language options:
    -c                  generate C code
    -c++                generate C++ code
    -c#                 generate C# code
    -java               generate Java code
    -xsd [<filename>]   generate XML schema definitions

  encoding rule options:
    -ber                generate BER encode/decode functions
    -cer                generate CER encode/decode functions
    -der                generate DER encode/decode functions
    -per                generate PER encode/decode functions
    -xer                generate XER encode/decode functions
    -xml                generate XML encode/decode functions

  basic options:
    -hfile <filename>   C or C++ header (.h) filename
                          (default is <ASN.1 Module Name>.h)
    -cfile <filename>   C or C++ source (.c or .cpp) filename
                          (default is <ASN.1 Module Name>.c)
    -print [<filename>] generate print functions
    -config <file>      specify configuration file
    -shortnames         reduce the length of compiler generated names
    -notypes            do not generate type definitions
    -nodecode           do not generate decode functions
    -noencode           do not generate encode functions
    -noIndefLen         do not generate indefinite length tests
```

```
  -noOpenExt          do not generate open extension elements
  -lax                do not generate constraint checks in code
  -compact            generate compact code
  -pdu <type>         designate <type> to be a Protocol Data Unit (PDU)
                      (<type> may be * to select all type definitions)
  -warnings           output compiler warning messages
  -o <directory>      set output file directory
  -I <directory>      set import file directory
  -trace              add trace diag msgs to generated code
  -list               generate listing
  -1990               parse 1990-compatible ASN.1 specification
                        (provides support for ROSE macro specs)
  -compat <version>   generate code compatible with previous
                      compiler version. <version> format is
                      x.x (for example, 5.3)

C/C++ options:
  -genBitMacros       generate named bit set, clear, test macros
  -genFree            generate memory free functions for all types
  -genMake            generate makefile to build generated code
  -maxlines [<num>]   set limit of number of lines per source file
                      (default value is 50000)
  -oh <directory>     set output directory for header files
  -static             generate static elements (not pointers)
  -w32                generate code for Windows O/S (default=GNU)

Java options:
  -pkgpfx <text>     Java package prefix
  -pkgname <text>    Java package name
  -dirs              output Java code to module name dirs
  -genjsources       generate <modulename>.mk for list of java files

C# options:
  -nspfx <text>      C# namespace prefix
  -namespace <text>  C# namespace name
  -dirs              output C# code to module name dirs
  -gencssources      generate <modulename>.mk for list of C# files

pro options:
  -events            generate code to invoke SAX-like event handlers
  -stream            generate stream-based encode/decode functions
  -tables            generate table constraint functions
  -strict            do strict checking of table constraint conformance
  -prtToStr [<filename>]
                     generate print-to-string functions (C/C++)
  -prtToStrm [<filename>]
                     generate print-to-stream functions (C/C++)
  -genTest  [<filename>]
                     generate sample test functions (C/C++)
  -reader            generate sample reader program(C/C++, BER/DER/PER)
  -writer            generate sample writer program(C/C++, BER/DER/PER)
```

```
  -compare [<filename>]
                    generate comparison functions (C/C++)
  -copy [<filename>] generate copy functions (C/C++)
  -maxcfiles        generate separate file for each function (C/C++)

XSD options:
  -tags             add ASN.1 tag info as appInfo in XSD spec
  -targetNS         specify XSD target namespace
  -noappinfo        do not include appInfo in generated XSD spec
```

Note that this usage summary shows all options for the pro version of ASN1C. Some of these options are not available in the basic version.

To use the compiler, at a minimum, an ASN.1 source file must be provided along with at least one set of encoding rules and a target output language. The source file specification can be a full pathname or only what is necessary to qualify the file. If directory information is not provided, the user's current default directory is assumed. If a file extension is not provided, the default extension ".asn" is appended to the name. Multiple source filenames may be specified on the command line to compile a set of files. The wildcard characters '*' and '%' are also allowed in source filenames (for example, the command 'asn1c *.asn' will compile all ASN.1 files in the current working directory).

The source file(s) must contain ASN.1 productions that define ASN.1 types and/or value specifications. This file must strictly adhere to the syntax specified in ASN.1 standard ITU X.680.. The -1990 command-line option should be used to parse files based on the 1990 ASN.1 standard (x.208) or that contain references to ROSE macro specifications..

The following table lists all of the command line options and what they are used for. The options are shown in alphabetical order. Note that the Java and C# options are not shown here. They are shown in their respective documents.

Option	Argument	Description
-1990	None	This option instructs the compiler to parse the older 1990 version of the ASN.1 syntax as specified in the now deprecated X.208 and X.209 standards. This syntax allowed the ANY construct as well as unnamed fields in SEQUENCE, SET, and CHOICE constructs. This option also allows for parsing and generation of code for ROSE OPERATION and ERROR macros and SNMP OBJECT-TYPE macros.
-ber	None	This option instructs the compiler to generate functions that implement the Basic Encoding Rules (BER) as specified in the X.690 ASN.1 standard.
-bitMacros	None	This option instructs the compiler to generate additional macros to set, clear, and test named bits in BIT STRING constructs. By default, only bit number constants are generated. Bit macros provide slightly better performance because mask values required to do the operations are computed at compile time rather than run-time.
-c	None	Generate C source code.
-c++ or -cpp	None	Generate C++ source code.

-c# or -csharp	None	Generate C# source code. See the *ASN1C C# User's Guide* for more information and options for generating C# code.
-cer	None	This option instructs the compiler to generate functions that implement the Canonical Encoding Rules (CER) as specified in the X.690 ASN.1 standard.
-cfile	<filename>	This option allows the specification of a C or C++ source (.c or .cpp) file to which all of the generated encode/decode functions will be written. If not specified, the default is to write to a series of .c or .cpp files based on the ASN.1 module name(s) of the documents being compiled.
-compact	None	This option instructs the compiler to generate more compact code at the expense of some constraint and error checking. This is an optimization option that should be used after an application is thoroughly tested.
-compat	<versionNumber>	Generate code compatible with an older version of the compiler. The compiler will attempt to generate code more closely aligned with the given previous release of the compiler. <versionNumber> is specified as x.x (for example, -compat 5.2)
-config	<filename>	This option is used to specify the name of a file containing configuration information for the source file being parsed. A full discussion of the contents of a configuration file is provided in the *Compiler Configuration File* section.
-der	None	This option instructs the compiler to generate functions that implement the Distinguished Encoding Rules (DER) as specified in the X.690 ASN.1 standard.
-events	None	Generate extra code to invoke user defined event and error handler callback methods (see the *Event Handlers* section). When combined with the *-notypes* option, causes pure parser functions to be generated.
-genCompare -compare	<filename>	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated compare functions will be written. Compare functions allow two variables of a given ASN.1 type to be compared for equality. The <filename> argument to this option is optional. If not specified, the functions will be written to <modulename>Compare.c where <modulename> is the name of the module from the ASN.1 source file.
-genCopy -copy	<filename>	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated copy functions will be written. Copy functions allow a copy to be made of an ASN1C generated variable. For C++, they cause copy constructors and assignment operators to be added to generated classes. The <filename> argument to this option is optional. If not specified, the functions will be written to <modulename>Copy.c where <modulename> is the name of the module from the ASN.1 source file.

-genFree	None	This option instructs the compiler to generate a memory free function for each ASN.1 production. Normally, memory is freed within ASN1C by using the *rtMemFree* run-time function to free all memory at once that is held by a context. Generated free functions allow finer grained control over memory freeing by just allowing the memory held for specific objects to be freed.
-genMake	None	This option instructs the compiler to generate a portable makefile for compiling the generated C or C++ code.
-genPrint -print	<filename>	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated print functions will be written. Print functions are debug functions that allow the contents of generated type variables to be written to stdout. The <filename> argument to this option is optional. If not specified, the print functions will be written to <modulename>Print.c where <modulename> is the name of the module from the ASN.1 source file.
-genPrtToStr -prtToStr	<filename>	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated "print-to-string" functions will be written. "Print-to-string" functions are similar to print functions except that the output is written to a user-provided text buffer instead of stdout. This makes it possible for the use to display the results on different output devices (for example, in a text window). The <filename> argument to this option is optional. If not specified, the functions will be written to <modulename>Print.c where <modulename> is the name of the module from the ASN.1 source file.
-genPrtToStrm -prtToStrm	<filename>	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated "print-to-stream" functions will be written. "Print-to-stream" functions are similar to print functions except that the output is written to a user-provided stream instead of stdout. The stream is in the form of an output callback function that can be set within the run-time context making it possible to redirect output to any type of device. The <filename> argument to this option is optional. If not specified, the functions will be written to <modulename>Print.c where <modulename> is the name of the module from the ASN.1 source file.
-genTables -tables	<filename>	This option is used to generate additional code for the handling of table constraints as defined in the X.682 standard. See <section> for additional details on the type of code generated to support table constraints.

-genTest	<filename>	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated "test" functions will be written. "Test" functions are used to populate an instance of a generated PDU type variable with random test data. This instance can then be used in an encode function call to test the encoder. Another advantage of these functions is that they can act as templates for writing your own population functions. The <filename> argument to this option is optional. If not specified, the functions will be written to <modulename>Test.c where <modulename> is the name of the module from the ASN.1 source file.
-hfile	<filename>	This option allows the specification of a header (.h) file to which all of the generated typedefs and function prototypes will be written. If not specified, the default is <modulename>.h where <modulename> is the name of the module from the ASN.1 source file.
-I	<directory>	This option is used to specify a directory that the compiler will search for ASN.1 source files for IMPORT items. Multiple –I qualifiers can be used to specify multiple directories to search.
-java	None	Generate Java source code. See the *ASN1C Java User's Guide* for more information on Java code generation.
-lax	None	This option instructs the compiler to not generate code to check constraints. When used in conjunction with the *–compact* option (described next), it produces the smallest code base for a given ASN.1 specification.
-list	None	Generate listing. This will dump the source code to the standard output device as it is parsed. This can be useful for finding parse errors.
-maxcfiles	None	Maximize number of generated C files. This option instructs the compiler to generate a separate .c file for each generated C function. In the case of C++, a separate .cpp file is generated for each control class, type, and C function. This is a space optimization option - it can lead to smaller executable sizes by allowing the linker to only link in the required program module object files.
-maxlines	<number>	This option is used to specify the maximum number of lines per generated .c or .cpp file. If this number is exceeded, a new file is started with a "_n" suffix where "n" is a sequential number. The default value if this is not specified is 50,000 lines which will prevent the VC++ "Maximum line numbers exceeded" warning that is common when compiling large ASN.1 source files. Note that this number is approximate - the next file will not be started until this number is exceeded and the compilation unit that is currently being generated is complete.
-noencode	None	This option suppresses the generation of encode functions.
-nodecode	None	This option suppresses the generation of decode functions.

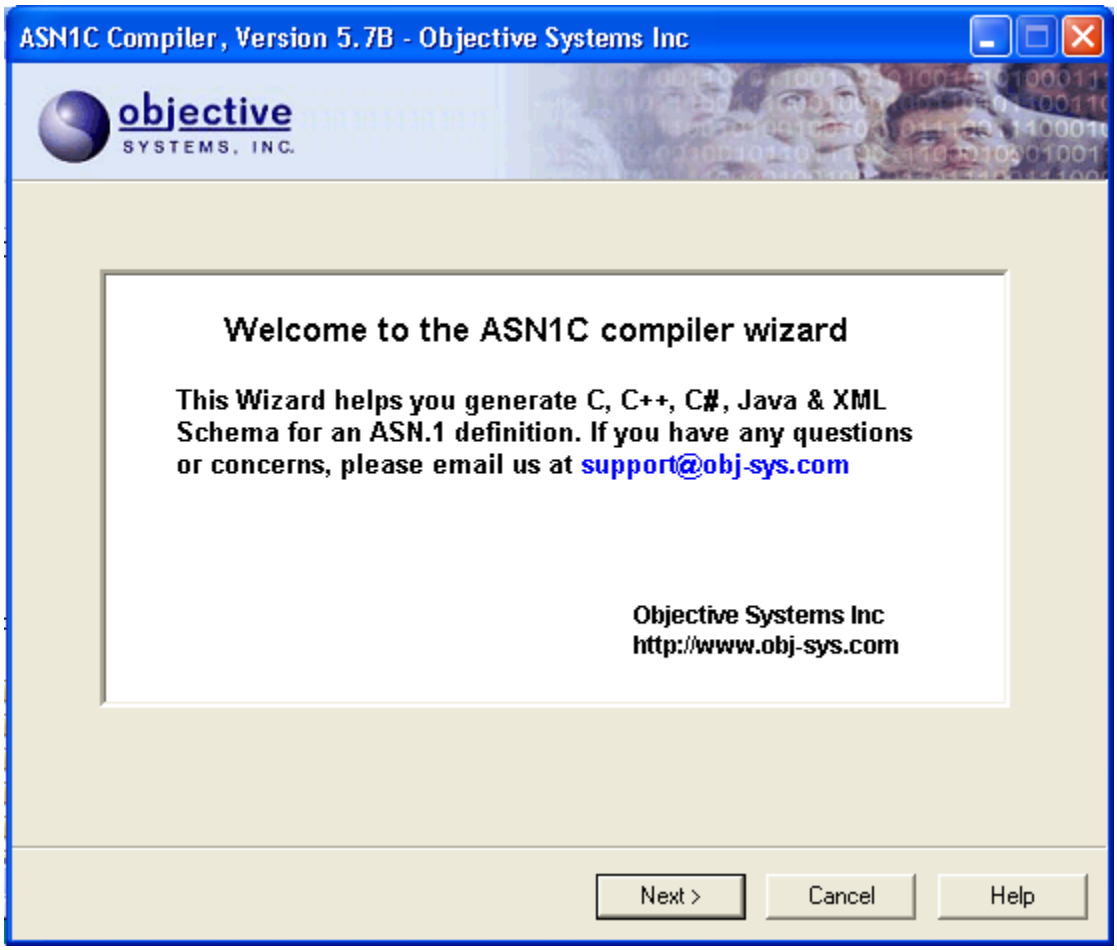
-noIndefLen	None	This option instructs the compiler to omit indefinite length tests in generated decode functions. These tests result in the generation of a large amount of code. If you know that your application only uses definite length encoding, this option can result in a much smaller code base size.
-noOpenExt	None	This option instructs the compiler to not add an open extension element (*extElem1*) in constructs that contain extensibility markers. The purpose of the element is to collect any unknown items in a message. If an application does not care about these unknown items, it can use this option to reduce the size of the generated code.
-notypes	None	This options suppresses the generation of type definitions. It is used in conjunction with the *-events* options to generate pure parser functions.
-o	<directory>	This option is used to specify the name of a directory to which all of the generated files will be written.
-oh	<directory>	This option is used to specify the name of a directory to which only the generated header files (*.H) will be written (C or C++ only).
-pdu	<typeName>	Designate given type name to be a PDU type. This will cause a C++ control class to be generated for the given type. By default, PDU types are determined to be types that are not referenced by any other types within a module. This option allows that behavior to be overridden. The '*' wildcard character may be specified for <typeName> to indicate that all productions within an ASN.1 module should be treated as PDU types.
-per	None	This option instructs the compiler to generate functions that implement the Packed Encoding Rules (PER) as specified in the ASN.1 standards.
-static	None	This has the same effect as specifying the global *<storage> static </storage>* configuration item. The compiler will insert static elements instead of pointer variables in generated structures.
-stream	None	This option instructs the compiler to generate stream-based encoders/ decoders instead of memory buffer based. This makes it possible to encode directly to or decode directly from a source or sink such as a file or socket. In the case of BER, it will also cause forward encoders to be generated, which will use indefinite lengths for all constructed elements in a message.
-strict	None	This option instructs the compiler to generate code for strict validation of table constraints. By default, generated code will not check for value field constraints. It should be noted that real world messages typically do not strictly follow value field table constraint definitions. Therefore, this option should be used with care.

-trace	None	This option is used to tell the compiler to add trace diagnostic messages to the generated code. These messages cause print statements to be added to the generated code to print entry and exit information into the generated functions. This is a debugging option that allows encode/decode problems to be isolated to a given production processing function. Once the code is debugged, this option should not be used as it adversely affects performance.
-warnings	None	Output information on compiler generated warnings.
-xer	None	This option instructs the compiler to generate functions that implement the XML Encoding Rules (XER) as specified in the X.693 ASN.1 standard.
-xml	None	This option instructs the compiler to generate functions that encode/decode data in an XML format that is more closely aligned with World-Wide Web Consortium (W3C) XML schema. The *-xsd* option can be used in conjunction with this option to generate a schema describing the XML format.
-xsd	<filename>	This option instructs the compiler to generate an equivalent XML Schema Definition (XSD) for each of the ASN.1 productions in the ASN.1 source file. The definitions are written to the given filename or to <modulename>.xsd if the filename argument is not provided.

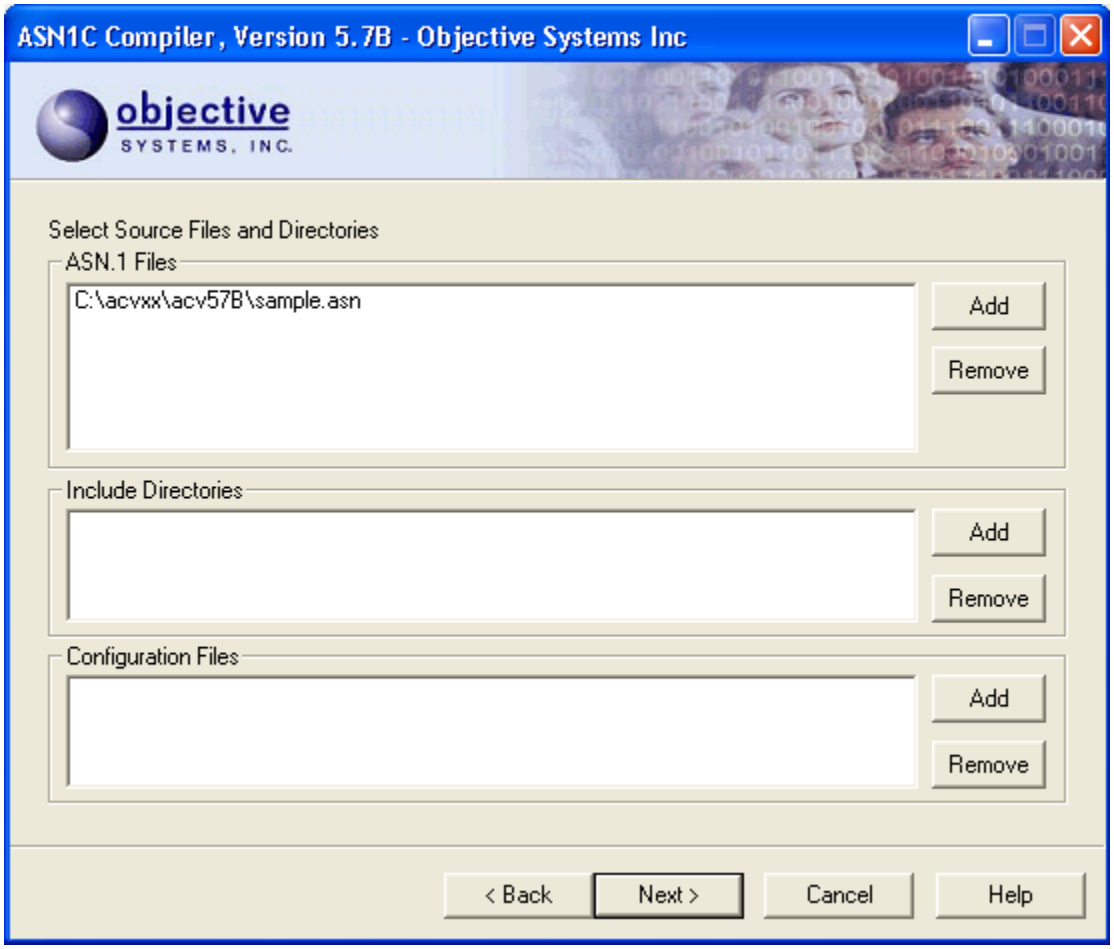
Using the GUI Wizard to Run ASN1C

The Windows version of ASN1C includes a graphical user interface (GUI) wizard that can be used as an alternative to the command-line version. This makes it possible to specify ASN.1 files and configuration files via file navigation windows, to set command line options by checking boxes, and to get online help on specific options.

The Windows installation program should have installed an 'ASN1C Compiler' option on your computer desktop and an 'ASN1C' option on the start menu. The wizard can be launched using either of these items. The following shows the initial Window that is displayed:


ASN1C Compiler, Version 5.7B - Objective Systems Inc
objective
SYSTEMS, INC.
Welcome to the ASN1C compiler wizard
This Wizard helps you generate C, C++, C#, Java & XML Schema for an ASN.1 definition. If you have any questions or concerns, please email us at support@obj-sys.com
Objective Systems Inc
http://www.obj-sys.com
Next >
Cancel
Help

This is a greeting window and no action is required. Click 'Next' to proceed to the next window:



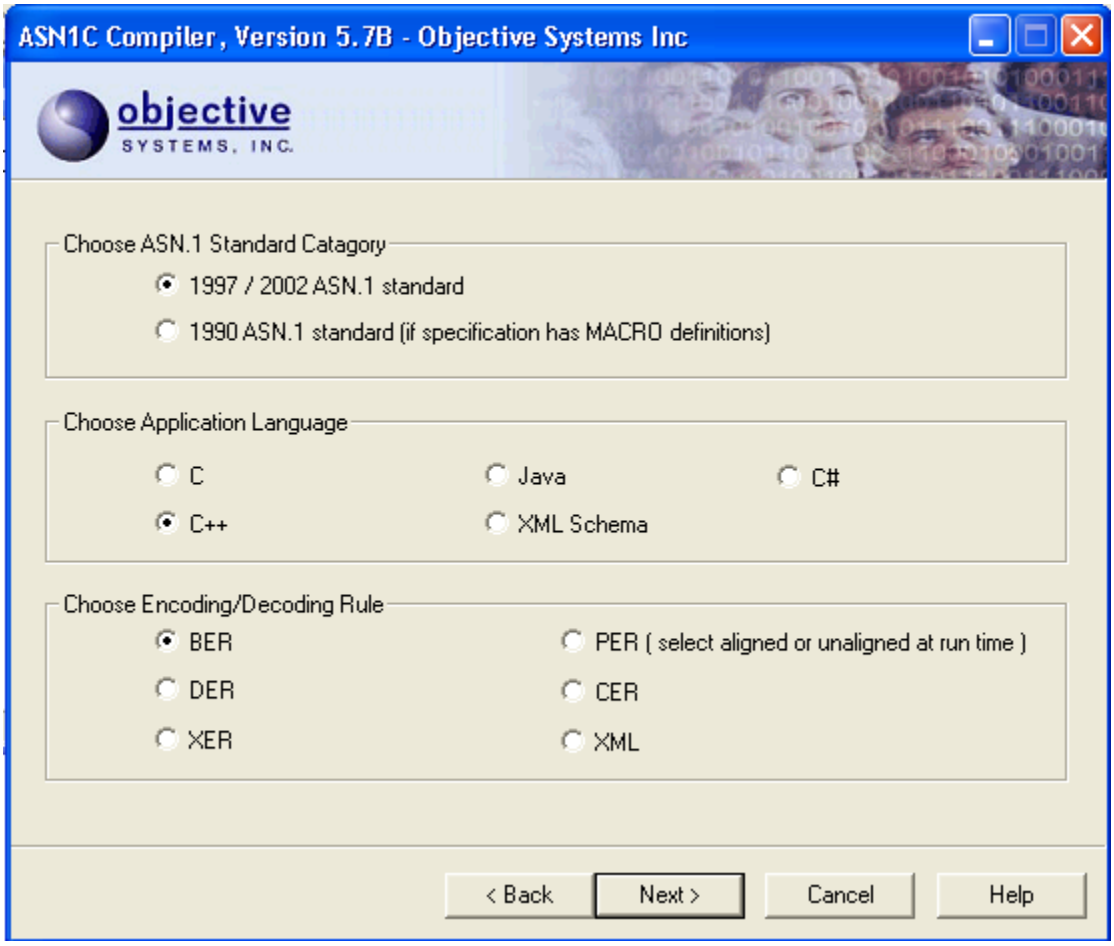
In this window, the ASN.1 file or files to be compiled are selected. This is done by clicking the "Add" button on the right hand side of the top windows pane. A file selection box will appear allowing you to select the ASN.1 files to be compiled. Files can be removed from the pane by highlighting the entry and clicking the "Remove" button.

Include directories are selected in a similar manner in the middle pane. These are directories the compiler will search for import files. By default, the compiler looks for files in the current working directory with the name of the module being imported and extension ".asn". Additional directories can be searched for these files by adding them here.

User defined configuration files are specified in the third pane. These allow further control of the compilation process. They are optional and are only needed if the default compilation process is to be altered (for example, if a type prefix is to be added to a generated type name). See the *Compiler Configuration File* section for details on defining these files.

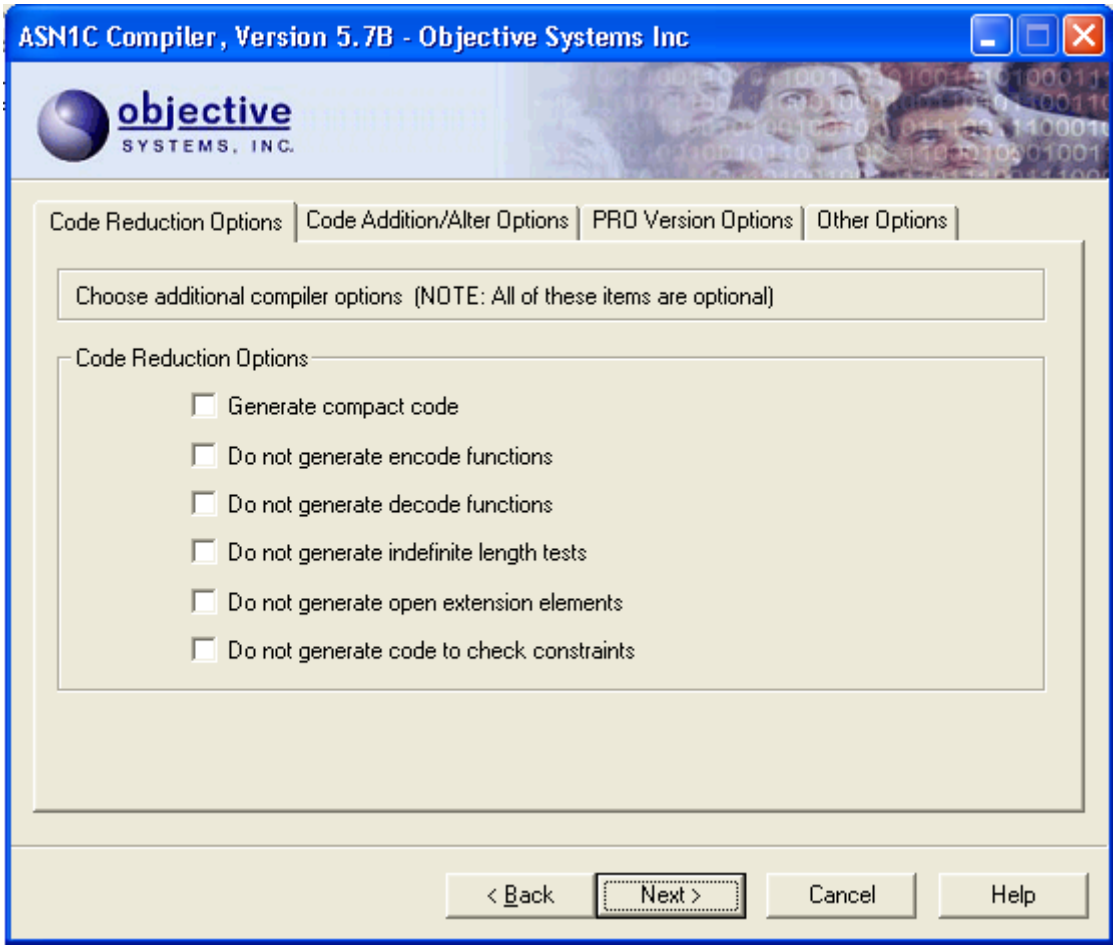
After specification of these files and directories is complete, click the 'Next' button to proceed to the next window.

The next window is as follows:



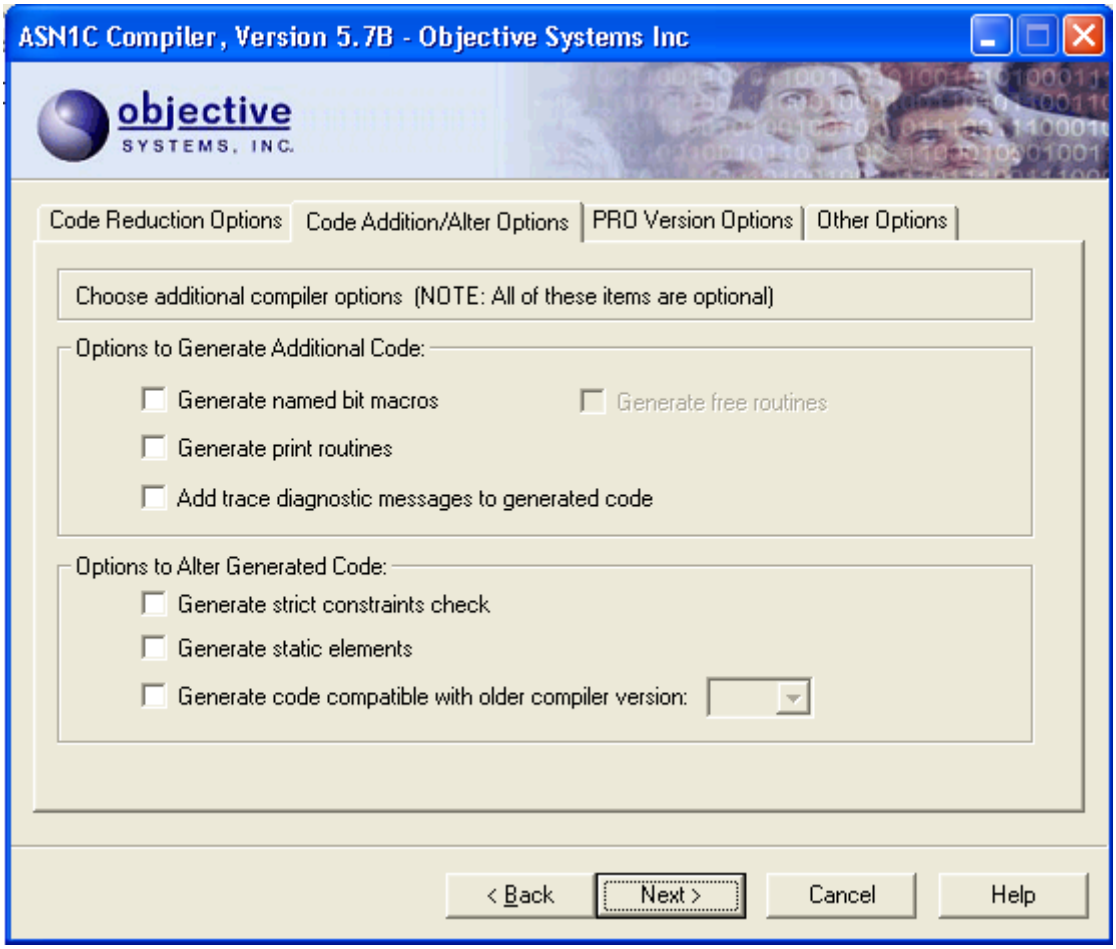
In this window, the encoding rules, target language, and ASN.1 syntax version (1990 or later) options are specified. Encoding rules are BER/DER, PER, XER or XML and can be specified using check boxes. This allows multiple sets of rules to be specified for a single compiler run. Target language and syntax version have radio button controls that only allow a single choice in each of these categories to be specified. It is required that at least one item from each of these categories be specified. Once these items have been specified, click 'Next' to proceed to the next window.

The next window is as follows:


ASN1C Compiler, Version 5.7B - Objective Systems Inc
objective
SYSTEMS, INC.
Code Reduction Options
Code Addition/Alter Options
PRO Version Options
Other Options
Choose additional compiler options (NOTE: All of these items are optional)
Code Reduction Options
Generate compact code
Do not generate encode functions
Do not generate decode functions
Do not generate indefinite length tests
Do not generate open extension elements
Do not generate code to check constraints
< Back
Next >
Cancel
Help

This dialog contains additional optional tabs that allow code reduction options, code addition/alter options, PRO version options and other options to be specified. Select the Code Reduction Options tab. These options reduce the amount of code generated at the expense of some features that are normally added to the code. Click on 'Help' to get a summary of what each of these options do or read the section *Running ASN1C from the Command-line*. All of these items are optional. Some items will be grayed-out if they are not applicable to the encoding rules / language combination selected on the previous page.

The Code Addition/Alter Options tab is as follows:


ASN1C Compiler, Version 5.7B - Objective Systems Inc
objective
SYSTEMS, INC.
Code Reduction Options
Code Addition/Alter Options
PRO Version Options
Other Options
Choose additional compiler options (NOTE: All of these items are optional)
Options to Generate Additional Code:
Generate named bit macros
Generate free routines
Generate print routines
Add trace diagnostic messages to generated code
Options to Alter Generated Code:
Generate strict constraints check
Generate static elements
Generate code compatible with older compiler version:
< Back
Next >
Cancel
Help

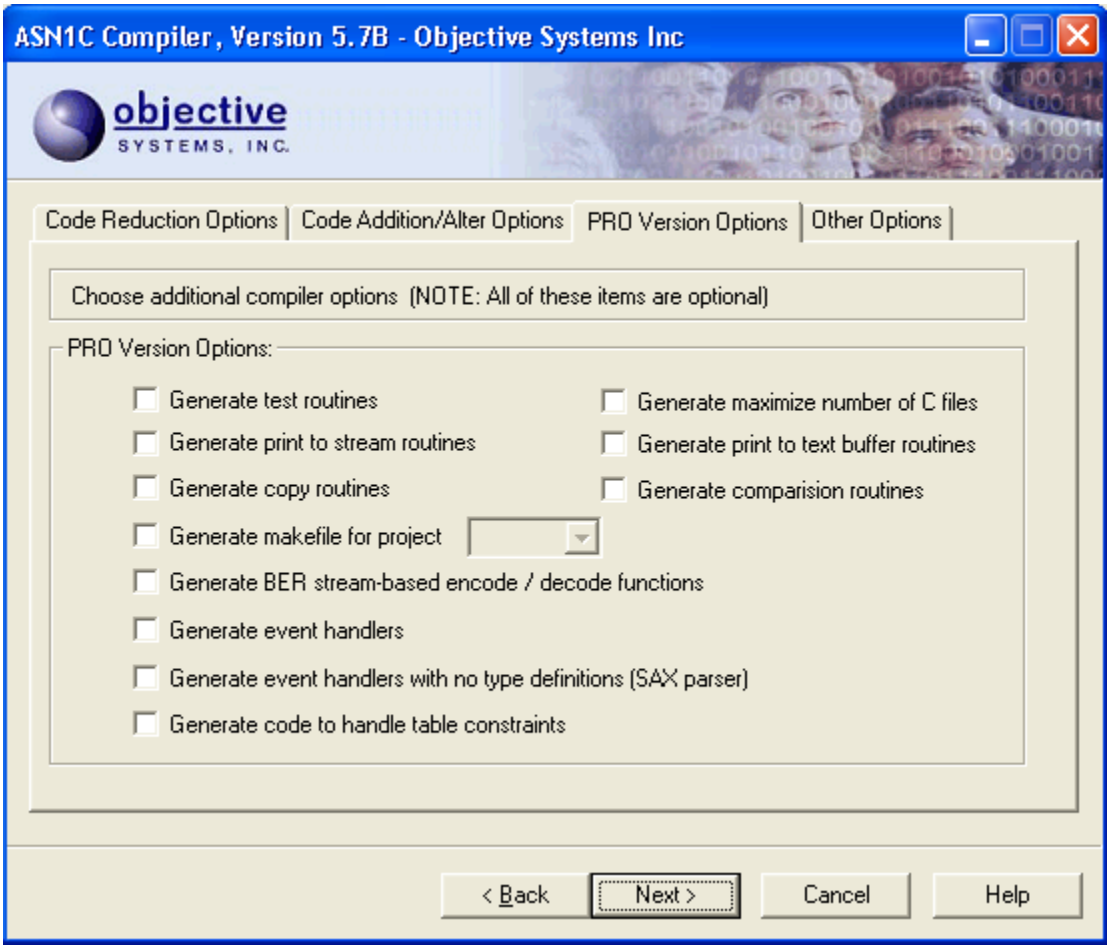
This tab contains additional optional items that allow additional code to be generated and that alters the type of code generated in some cases to be specified. Named bit macros control the generation of special macros for setting, clearing and testing bits in BIT STRING types. Print routines are special functions for printing the contents of generated type variables. Trace diagnostic messages are added to encode/decode functions to trace the sequence of function calls used in the encoding or decoding of a message.

The 'Generate static elements' option is used to add static elements to CHOICE constructs instead of pointer values.

The 'Generate code compatible ...' option is for generating code compatible with an older release of the compiler.

Click on 'Help' to get a summary of what each of these options do or read the section *Running ASN1C from the Command-line*. All of these items are optional. Some items will be grayed-out if they are not applicable to the encoding rules / language combination selected on the previous page.

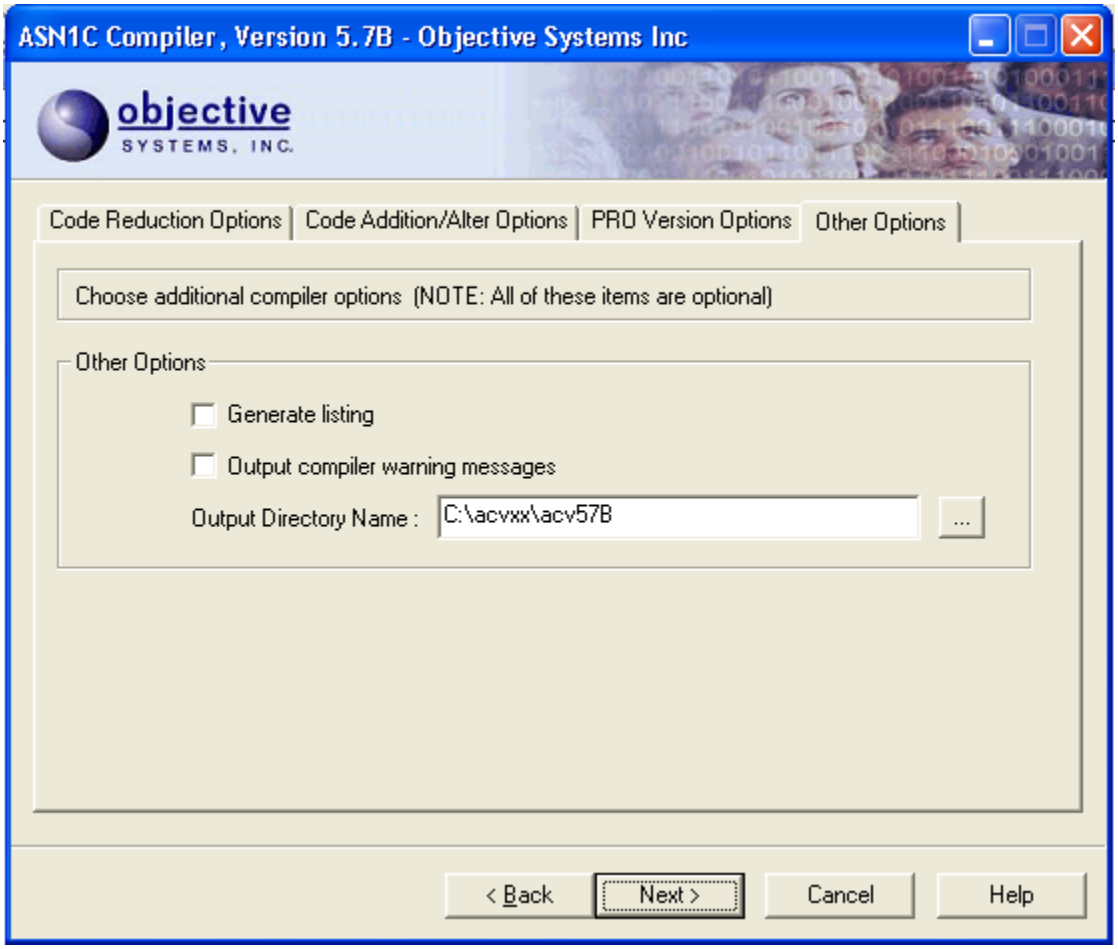
The PRO version Options tab is as follows:


ASN1C Compiler, Version 5.7B - Objective Systems Inc
objective
SYSTEMS, INC.
Code Reduction Options
Code Addition/Alter Options
PRO Version Options
Other Options
Choose additional compiler options (NOTE: All of these items are optional)
PRO Version Options:
Generate test routines
Generate maximize number of C files
Generate print to stream routines
Generate print to text buffer routines
Generate copy routines
Generate comparision routines
Generate makefile for project
Generate BER stream-based encode / decode functions
Generate event handlers
Generate event handlers with no type definitions (SAX parser)
Generate code to handle table constraints
< Back
Next >
Cancel
Help

This allows options to be set that are only present in the professional version of the compiler.

Click on 'Help' to get a summary of what each of these options do. All of these items are optional. Some items will be grayed-out if they are not applicable to the encoding rules / language combination selected on the previous page.

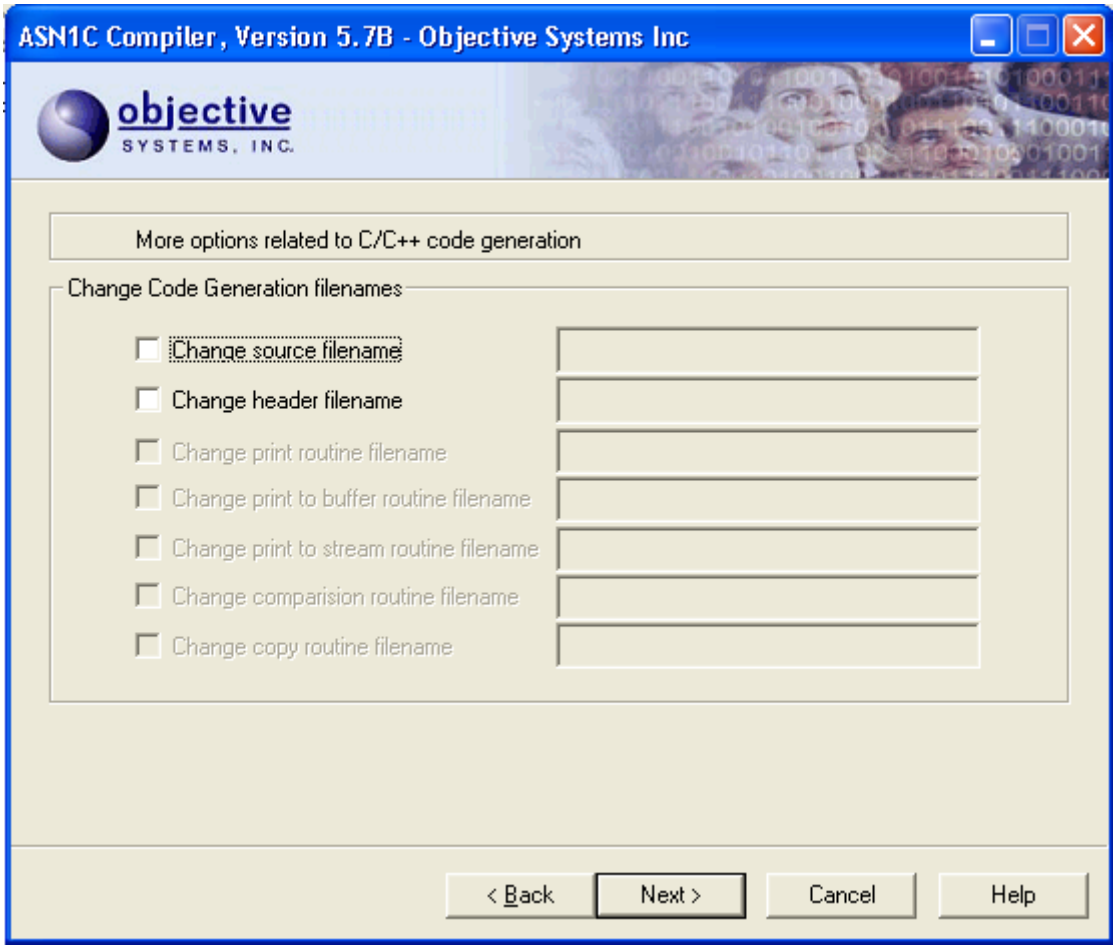
The Other Options tab is as follows:



The 'Output Directory Name' control at the bottom of the page allows you to specify where your generated code will be stored. By default, this is the same directory as the ASN.1 source files. A different output can be selected by either typing the name directly into the box or clicking the "[…]" button to open a file navigation window.

Once specification of these items is complete, click 'Next' to move to the next window.

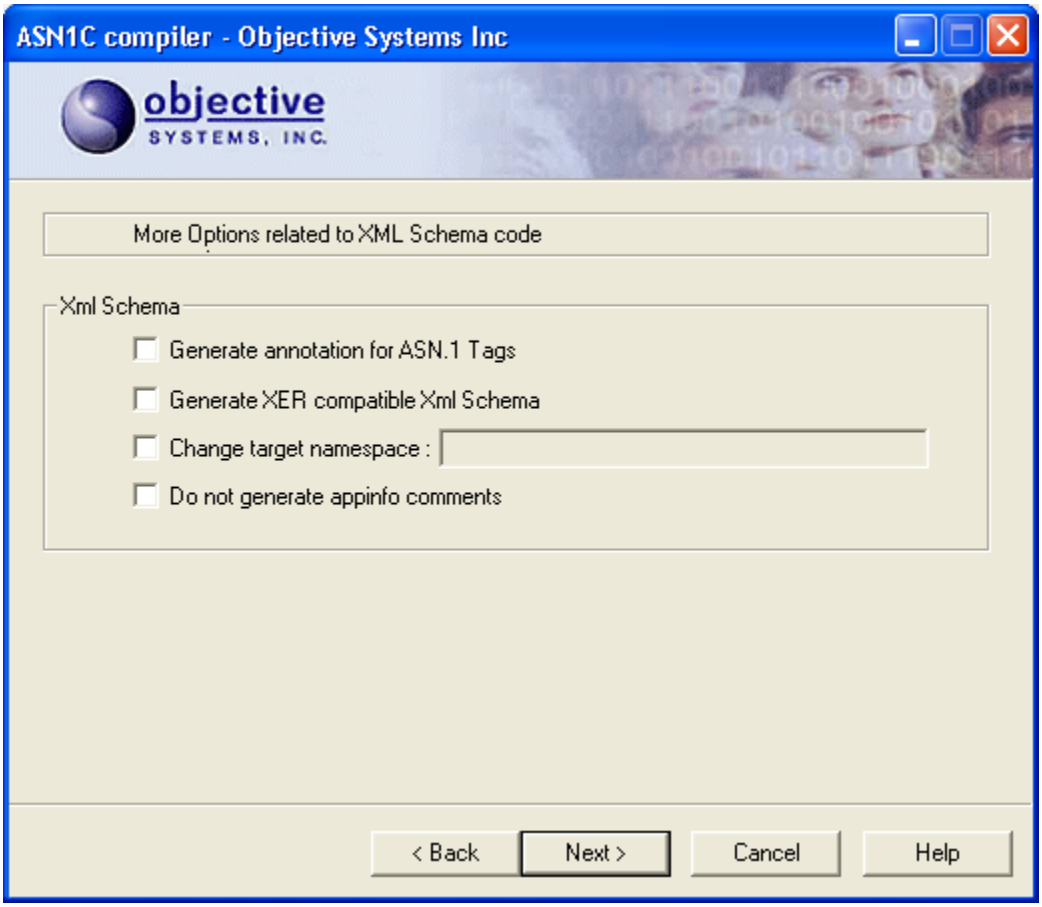
The next window is as follows:



This window is specific to the target language (C/C++, Java, or XML Schema) that was selected earlier. The window shown here is for C/C++. If Java or XML Schema was selected, a different window will be presented allowing selection of options specific to those languages.

The C/C++ window allows additional code generation options and output filenames to be chosen. The output filenames are optional; the compiler will create default names using the name of the ASN.1 modules being compiled. Once again, options may be grayed out based on selections made on previous pages. Click 'Next' to move to the next window.

If XML Schema was selected as the target language, the following window would be displayed:

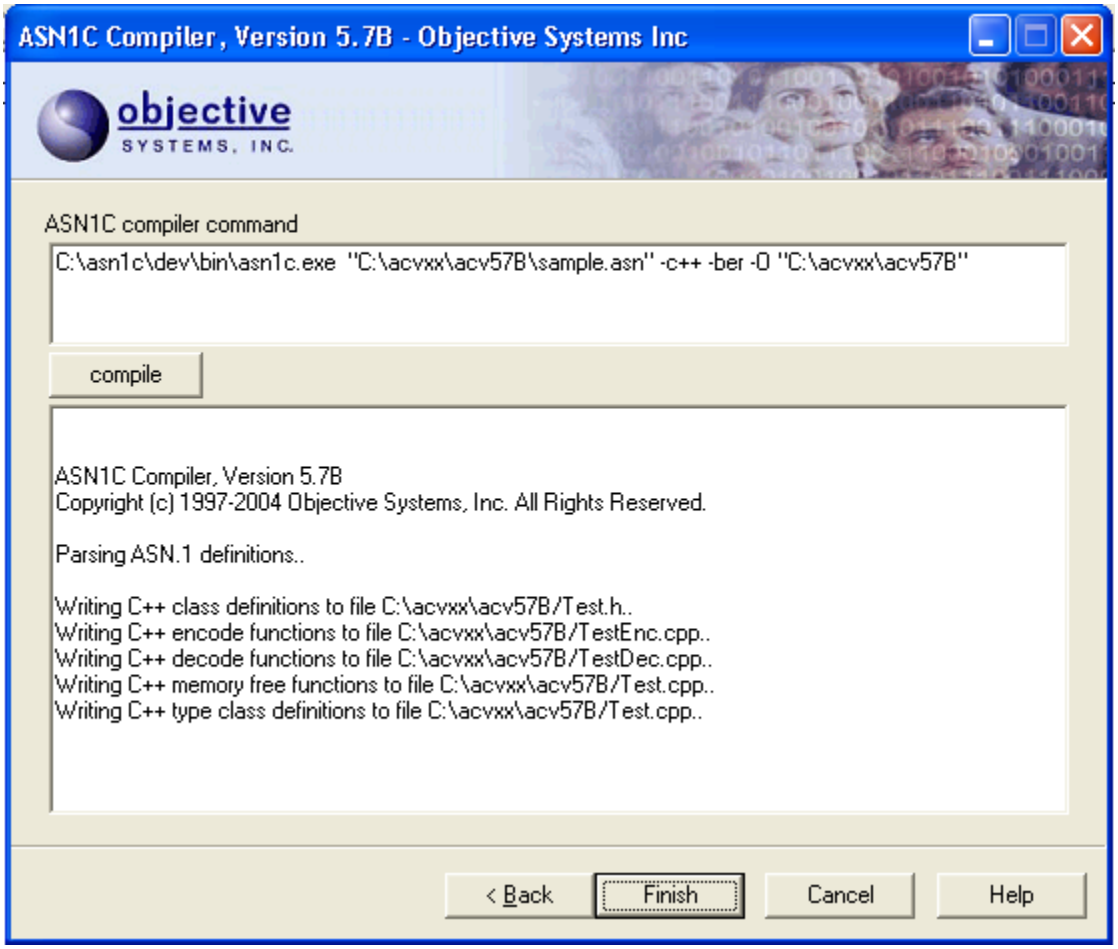


This allows options specific to XML Schema to be selected. The user can specify that annotation application information (<appinfo>) be added for ASN.1 tag values. This application information can be used by an XML data binding application to generate BER or DER encoders/decoders directly from the schema definition.

The user can also specify that the generated schema be compatible with the ASN.1 XML encoding rules (XER). By default, the generated schema is compatible with the standard XML schema types.

Once complete, click 'Next' to proceed to the final page.

The final window is as follows:


ASN1C Compiler, Version 5.7B - Objective Systems Inc
objective
SYSTEMS, INC.
ASN1C compiler command
C:\asn1c\dev\bin\asn1c.exe "C:\acvxx\acv57B\sample.asn" -c++ -ber -O "C:\acvxx\acv57B"
compile
ASN1C Compiler, Version 5.7B
Copyright (c) 1997-2004 Objective Systems, Inc. All Rights Reserved.
Parsing ASN.1 definitions..
Writing C++ class definitions to file C:\acvxx\acv57B/Test.h..
Writing C++ encode functions to file C:\acvxx\acv57B/TestEnc.cpp..
Writing C++ decode functions to file C:\acvxx\acv57B/TestDec.cpp..
Writing C++ memory free functions to file C:\acvxx\acv57B/Test.cpp..
Writing C++ type class definitions to file C:\acvxx\acv57B/Test.cpp..
< Back
Finish
Cancel
Help

As can be seen, the GUI has constructed an ASN1C command-line using the specified options. The user can now click 'Compile' to execute the compilation. The 'Back' button can be used to step back through the sequence of screens in order to change any of the previous inputs.

After 'Compile' is clicked, the results of the compilation are shown in the results text window. If the compilation was successful, the generated code is written to the files in the given output directory.

Clicking the "Finish" or "Cancel" button will terminate the GUI wizard.

Compiling and Linking Generated Code

C/C++ source code generated by the compiler can be compiled using any ANSI standard C or C++ compiler. The only additional option that must be set is the inclusion of the ASN.1 C/C++ header file include directory with the –I option.

When linking a program with compiler-generated code, it is necessary to include the ASN.1 run-time libraries. It is necessary to include at least one of the encoding rules libraries (asn1ber, asn1per, or asn1xer) as well as the common run-time functions library (asn1rt). See the *ASN1C C/C++ Run-time Reference Manual* for further details on these libraries.

For static linking on Windows systems, the name of the library files are *asn1ber_a.lib*, *asn1per_a.lib*, or *asn1xer_a.lib* for BER/DER, PER, or XER respectively, and *asn1rt_a.lib* for the common run-time components. On UNIX/Linux, the library names are *libasn1ber.a, libasn1per.a, libasn1xer.a*, and *libasn1rt.a*. The library files are located in the lib subdirectory. For UNIX, the –L switch should be used to point to the subdirectory path and –lasn1ber, -lasn1per, –lasn1xer, and/or –lasn1rt used to link with the libraries. For Windows, the –LIBPATH switch should be used to specify the library path.

There are several other variations of the C/C++ run-time library files for Windows. The following table summarizes what options were used to build each of these variations:

Library Files	Description
asn1rt_a.lib *asn1ber_a.lib* *asn1per_a.lib* *asn1xer_a.lib*	Static single-threaded libraries. These are built without –MT (multithreading) and –MD (dynamic link libraries) options. These are not thread-safe. However, they provide the smallest footprint of the different libraries.
asn1rt.lib *asn1ber.lib* *asn1per.lib* *asn1xer.lib*	DLL libraries. These are used to link against the DLL versions of the run-time libraries (*asn1rt.dll*, etc.)
asn1rtmt_a.lib *asn1bermt_a.lib* *asn1permt_a.lib* *asn1xermt_a.lib*	Static multi-threaded libraries. These libraries were built with the –MT option. They should be used if your application contains threads and you wish to link with the static libraries (note: the DLL's are also thread-safe).
asn1rtmd_a.lib *asn1bermd_a.lib* *asn1permd_a.lib* *asn1xermd_a.lib*	DLL-ready multi-threaded libraries. These libraries were built with the –MD option. They allow linking additional object modules in with the ASN1C run-time modules to produce larger DLL's.

For dynamic linking on UNIX/Linux, a shared object version of each run-time library is included in the lib subdirectory. This file typically has the extension .so (for shared object) or .sl (for shared library). See the documentation for your UNIX compiler to determine how to link using these files (it varies for different types of UNIX systems).

Compiling and linking code generated to support the XML encoding rules (XER) is more complex then the other rules because XER requires the use of third-party XML parser software. This requires the use of additional include directories when compiling and libraries when linking. The C++ sample programs that are provided use the EXPAT XML parser (http://www.expat.org/). All of the necessary include files and binary libraries are included with the distribution for using this parser. If a different parser is to be used, consult the vendor's documentation for compile and link procedures.

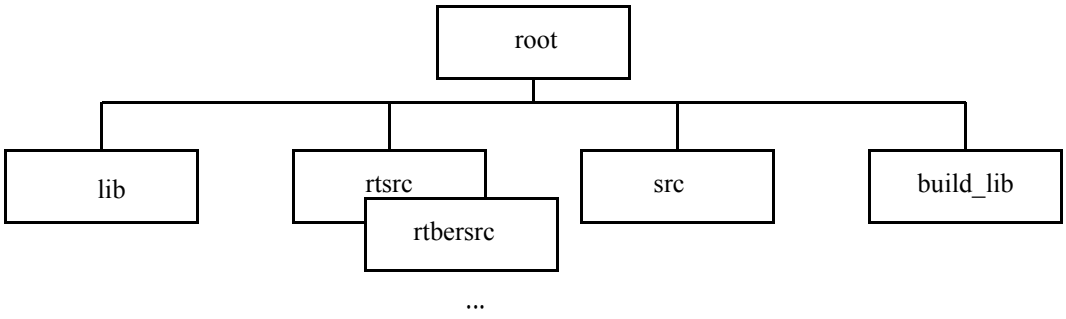
See the *makefile* in any of the sample subdirectories of the distribution for an example of what must be included to build a program using generated source code.

Porting Run-time Code to Other Platforms

The run-time source version of the compiler includes ANSI-standard source code for the base run-time libraries. This code can be used to build binary versions of the run-time libraries for other operating environments. Included with the source code is a portable makefile that can be used to build the libraries on the target platform with minimal changes. All platform-specific items are isolated in the platform.mk file in the root directory of the installation.

The procedure to port the run-time code to a different platform is as follows (note: this assumes common UNIX or GNU compilation utilities are in place on the target platform).

1. Create a directory tree containing a root directory (the name does not matter) and lib, src, rt*src, and build_lib subdirectories (note: in these definitions, * is a wildcard character indicating there are multiple directories matching this pattern). The tree should be as follows:



2. Copy the files ending in extension ".mk" from the root directory of the installation to the root directory of the target platform (note: if going from DOS to UNIX or vice-versa, FTP the files in ASCII mode to ensure lines are terminated properly).

3. Copy all files from the src and the different rt*src subdirectories from the installation to the src and rt*src directories on the target platform (note: if going from DOS to UNIX or vice-versa, FTP the files in ASCII mode to ensure lines are terminated properly).

4. Copy the makefile from the build_lib subdirectory of the installation to the build_lib subdirectory on the target platform (note: if going from DOS to UNIX or vice-versa, FTP the files in ASCII mode to ensure lines are terminated properly).

5. Edit the **platform.mk** file in the root subdirectory and modify the compilation parameters to fit those of the compiler of the target system. In general, the following parameters will need to be adjusted:

   CC C compiler executable name
   CCCC++ compiler executable name
   CFLAGS_Flags that should be specified on the C or C++ command line

   The platform.w32 and platform.gnu files in the root directory of the installation are sample files for Windows 32 (Visual C++) and GNU compilers respectively. Either of these can be renamed to platform.mk for building in either of these environments.

6. Invoke the makefile in the build_lib subdirectory.

If all parameters were set up correctly, the result should be binary library files created in the lib subdirectory.

Compiler Configuration File

In addition to command line options, a configuration file can be used to specify compiler options. These options can be applied not only globally but also to specific modules and productions.

A simple form of the Extended Markup Language (XML) is used to format items in the file. This language was chosen because it is fairly well known and provides a natural interface for representing hierarchical data such as the structure of ASN.1 modules and productions. The use of an external configuration file was chosen over embedding directives within the ASN.1 source itself due to the fact that ASN.1 source versions tend to change frequently. An external configuration file can be reused with a new version of an ASN.1 module, but internal directives would have to be reapplied to the new version of the ASN.1 code.

At the outer level of the markup is the <asn1config> </asn1config> tag pair. Within this tag pair, the specification of global items and modules can be made. Global items are applied to all items in all modules. An example would be the <storage> qualifier. A storage class such as dynamic can be specified and applied to all productions in all modules. This will cause dynamic storage (pointers) to be used for any embedded structures within all of the generated code to reduce memory consumption demands.

The specification of a module is done using the <module></module> tag pair. This tag pair can only be nested within the top-level <asn1config> section. The module is identified by using the required <name></name> tag pair or by specifying the name as an attribute (for example, <module name=”MyModule”>). Other attributes specified within the <module> section apply only to that module and not to other modules specified within the specification. A complete list of all module attributes is provided in the table at the end of this section.

The specification of an individual production is done using the <production></production> tag pair. This tag pair can only be nested within a <module> section. The production is identified by using the required <name></name> tag pair or by specifying the name as an attribute (for example, <production name=”MyProd”>). Other attributes within the production section apply only to the referenced production and nothing else. A complete list of attributes that can be applied to individual productions is provided in the table at the end of this section.

When an attribute is specified in more than one section, the most specific application is always used. For example, assume a <typePrefix> qualifier is used within a module specification to specify a prefix for all generated types in the module and another one is used to a specify a prefix for a single production. The production with the type prefix will be generated with the type prefix assigned to it and all other generated types will contain the type prefix assigned at the module level.

Values in the different sections can be specified in one of the following ways:

1. Using the <name>value</name> form. This assigns the given value to the given name. For example, the following would be used to specify the name of the “H323-MESSAGES” module in a module section:

   <name>H323-MESSAGES</name>

2. Flag variables that turn some attribute on or off would be specified using a single <name/> entry. For example, to specify a given production is a PDU, the following would be specified in a production section:

   <isPDU/>

3. An attribute list can be associated with some items. This is normally used as a shorthand form for specifying lists of names. For example, to specify a list of type names to be included in the generated code for a particular module, the following would be used:

```
<include types="TypeName1,TypeName2,TypeName3"/>
```

The following are some examples of configuration specifications:

```
<asn1config><storage>dynamic</storage></asn1config>
```

This specification indicates dynamic storage should be used in all places where its use would result in significant memory usage savings within all modules in the specified source file.

```
<asn1config>
  <module>
    <name>H323-MESSAGES</name>
    <sourceFile>h225.asn</sourceFile>
    <typePrefix>H225</typePrefix>
  </module>
  …
</asn1config>
```

This specification applies to module 'H323-MESSAGES' in the source file being processed. For IMPORT statements involving this module, it indicates that the source file 'h225.asn' should be searched for specifications. It also indicates that when C or C++ types are generated, they should be prefixed with 'H225'. This can help prevent name clashes if one or more modules are involved and they contain productions with common names.

The following tables specify the list of attributes that can be applied at all of the different levels: global, module, and individual production:

Global Level

These attributes can be applied at the global level by including them within the <asn1config> section:

Name	Values	Description

<storage></storage>	*dynamic*, *static*, *list*, *array*, or *dynamicArray* keyword.	If *dynamic*, it indicates that dynamic storage (i.e., pointers) should be used everywhere within the generated types where use could result in lower memory consumption. These places include the array element for sized SEQUENCE OF/SET OF types and all alternative elements within CHOICE constructs. If *static*, it indicates static type should be used in these places. In general, static types are easier to work with. If *list*, a linked-list type will be used for SEQUENCE OF/SET OF constructs instead of an array type. If *array*, an array type will be used for SEQUENCE OF/SET OF constructs. The *maxSize* attribute can be used in this case of specify the size of the array variable (for example, <storage maxSize=”12”> array </storage>). If *dynamicArray*, a dynamic array will be used for SEQUENCE OF/ SET OF constructs. A dynamic array is an array that uses dynamic storage for the array elements.

Module Level

These attributes can be applied at the module level by including them within a <module> section:

Name	Values	Description
<name> </name>	module name	This attribute identifies the module to which this section applies. It is required.
<include types=”names” values=”names”/>	ASN.1 type or values names are specified as an attribute list	This item allows a list of ASN.1 types and/or values to be included in the generated code. By default, the compiler generates code for all types and values within a specification. This allows the user to reduce the size of the generated code base by selecting only a subset of the types/values in a specification for compilation. Note that if a type or value is included that has dependent types or values (for example, the element types in a SEQUENCE, SET, or CHOICE), all of the dependent types will be automatically included as well.
<include importsFrom= ”name” />	ASN.1 module name(s) specified as an attribute list.	This form of the include directive tells the compiler to only include types and/or values in the generated code that are imported by the given module(s).
<exclude types=”names” values=”names”/>	ASN.1 type or values names are specified as an attribute list	This item allows a list of ASN.1 types and/or values to be excluded in the generated code. By default, the compiler generates code for all types and values within a specification. This is generally not as useful as in *include* directive because most types in a specification are referenced by other types. If an attempt is made to exclude a type or value referenced by another item, the directive will be ignored.
<storage> </storage>	*dynamic*, *static*, *list*, *array*, or *dynamicArray* keyword.	The definition is the same as for the global case except that the specified storage type will only be applied to generated C and C++ types from the given module.
<sourceFile> </sourceFile>	source file name	Indicates the given module is contained within the given ASN.1 source file. This is used on IMPORTs to instruct the compiler where to look for imported definitions. This replaces the **module.txt** file used in previous versions of the compiler to accomplish this function.
<typePrefix> </typePrefix>	prefix text	This is used to specify a prefix that will be applied to all generated C and C++ typedef names (note: for C++, the prefix is applied after the standard ‘ASN1T_’ prefix). This can be used to prevent name clashes if multiple modules are involved in a compilation and they all contain common names.

<enumPrefix> </enumPrefix>	prefix text	This is used to specify a prefix that will be applied to all generated enumerated identifiers within a module. This can be used to prevent name clashes if multiple modules are involved in a compilation. (note: this attribute is normally not needed for C++ enumerated identifiers because they are already wrapped in a structure to allows the type name to be used as an additional identifier).
<valuePrefix> </valuePrefix>	prefix text	This is used to specify a prefix that will be applied to all generated value constants within a module. This can be used to prevent name clashes if multiple modules are involved that use a common name for two or more different value declarations.
<noPDU/>	n/a	Indicates that this module contains no PDU definitions. This is normally true in modules that are imported to get common type definitions (for example, InformationFramework). This will prevent the C++ version of the compiler from generating any control class definitions for the types in the module.
<intCType>	byte, int16, uint16, int32, uint32, int64, string	This is used to specify a specific C integer type be used for all unconstrained integer type. By default, ASN1C will use the int32 (32-bit integer) type for all unconstrained integers.
<arcCType>	int32, int64	The is used to specify a specific C integer type be used for the arc types in Object Identifier definitions. By default, int32 (32-bit integer arc values) are generated.
<namespace> </namespace>	namespace name	This is used to specify the namespace name for the given module. By default, Asn1c compiler will use the “http://www.obj-sys.com” as the module namespace. This option should be used with XML encoding rule (XML) only. Asn1c compiler will ignore this option usage with other encoding rules.

Production Level

These attributes can be applied at the production level by including them within a <production> section:

Name	Values	Description
<name> </name>	module name	This attribute identifies the module to which this section applies. It is required.
<ctype>	byte, int16, uint16, int32, uint32, int64, string	This is used to specify a specific C integer type be used in place of the default integer definition generated by ASN1C. In general, ASN1C will try and use the smallest integer type available based on the value or value range constraint on the integer type. If the integer is not constrained, the int32 (32-bit integer) type will be used.

<storage> </storage>	*dynamic*, *static*, *list*, *array*, or *dynamicArray* keyword.	The definition is the same as for the global case except that the specified storage type will only be applied to the generated C or C++ type for the given production.
<typePrefix> </typePrefix>	prefix text	This is used to specify a prefix that will be applied to all generated C and C++ typedef names (note: for C++, the prefix is applied after the standard 'ASN1T_' prefix). This can be used to prevent name clashes if multiple modules are involved in a compilation and they all contain common names.
<enumPrefix> </enumPrefix>	prefix text	This is used to specify a prefix that will be applied to all generated enumerated identifiers within a module. This can be used to prevent name clashes if multiple modules are involved in a compilation. (note: this attribute is normally not needed for C++ enumerated identifiers because they are already wrapped in a structure to allows the type name to be used as an additional identifier).
<isBigInteger/>	n/a	This is a flag variable (an 'empty element' in XML terminology) that specifies that this production will be used to store an integer larger than the C or C++ int type on the given system (normally 32 bits). A C string type (char*) will be used to hold a textual representation of the value. This qualifier can be applied to either an integer or constructed type. If constructed, all integer elements within the constructed type are flagged as big integers.
<isPDU/>	n/a	This is a flag variable that specifies that this production represents a Protocol Data Unit (PDU). This is defined as a production that will be encoded or decoded from within the application code. This attribute only makes a difference in the generation of C++ classes. Control classes that are only used in the application code are only generated for types with this attribute set.

Element Level

These attributes can be applied at the element level by including them within an <element> section:

Name	**Values**	**Description**
<name> </name>	element name	This attribute identifies the element within a SEQUENCE, SET, or CHOICE construct to which this section applies. It is required.
<isOpenType/>	n/a	This is a flag variable (an 'empty element' in XML terminology) that specifies that this element will be decoded as an open type (i.e. skipped). Refer to the section on deferred decoding for further information.

Compiler Error Reporting

Errors that can occur when generating source code from an ASN.1 source specification take two forms: syntax errors and semantics errors.

Syntax errors are errors in the ASN.1 source specification itself. These occur when the rules specified in the ASN.1 grammar are not followed. ASN1C will flag these types of errors with the error message 'Syntax Error' and abort compilation on the source file. The offending line number will be provided. The user can re-run the compilation with the '-l' flag specified to see the lines listed as they are parsed. This can be quite helpful in tracking down a syntax error.

The most common types of syntax errors are as follows:

- Invalid case on identifiers: module name must begin with an uppercase letter, productions (types) must begin with an uppercase letter, and element names within constructors (SEQUENCE, SET, CHOICE) must begin with lowercase letters.

- Elements within constructors not properly delimited with commas: either a comma is omitted at the end of an element declaration, or an extra comma is added at the end of an element declaration before the closing brace.

- Invalid special characters: only letters, numbers, and the hyphen (-) character are allowed. C programmers tend to like to use the underscore character (_) in identifiers. This is not allowed in ASN.1. Conversely, C does not allow hyphens in identifiers. To get around this problem, ASN1C converts all hyphens in an ASN.1 specification to underscore characters in the generated code.

Semantics errors occur on the compiler back-end as the code is being generated. In this case, parsing was successful, but the compiler does not know how to generate the code. These errors are flagged by embedding error messages directly in the generated code. The error messages always begin with an identifier with the prefix '%ASN-', so a search can be done for this string in order to find the locations of the errors. A single error message is output to stderr after compilation on the unit is complete to indicate error conditions exist.

Generated C/C++ Source Code

Header (.h) File

The generated C or C++ include file contains a section for each ASN.1 production defined in the ASN.1 source file. Different items will be generated depending on whether the selected output code is C or C++. In general, C++ will add some additional items (such as a control class definition) onto what is generated for C.

The following items are generated for each ASN.1 production:

- Tag value constant
- Choice tag constants (CHOICE type only)
- Named bit number constants (BIT STRING type only)
- Enumerated type option values (ENUMERATED or INTEGER type only)
- C type definition
- Encode function prototype
- Decode function prototype
- Other function prototypes depending on selected options (for example, print)
- C++ control class definition (C++ only)

A sample section from a C header file is as follows:

```
/**************************************************************/
/*                                                            */
/*  EmployeeNumber                                            */
/*                                                            */
/**************************************************************/

#define TV_EmployeeNumber(TM_APPL|TM_PRIM|2)

typedef ASN1INT EmployeeNumber;

EXTERN int asn1E_EmployeeNumber (ASN1CTXT* pctxt,
   EmployeeNumber *pvalue, ASN1TagType tagging);

EXTERN int asn1D_EmployeeNumber (ASN1CTXT* pctxt,
   EmployeeNumber *pvalue, ASN1TagType tagging, int length);
```

This corresponds to the following ASN.1 production specification:

```
EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER
```

In this definition, TV_EmployeeNumber is the tag constant. Doing a logical OR on the class, form, and identifier fields forms this constant. This constant can be used in a comparison operation with a tag parsed from a message.

The following line:

```
typedef ASN1INT EmployeeNumber;
```

declares EmployeeNumber to be of an integer type (note: ASN1INT and other primitive type definitions can be found in the asn1type.h header file).

asn1E_EmployeeNumber and *asn1D_EmployeeNumber* are function prototypes for the encode and decode functions respectively. These are BER function prototypes. If the –per switch is used, PER function prototypes are generated. The PER prototypes begin with the prefix *asn1PE_* and *asn1PD_* for encoder and decoder respectively. XER function prototypes begin with *asn1XE_* and *asn1XD_*.

A sample section from a C++ header file for the same production is as follows:

```
/************************************************************/
/*                                                          */
/*  EmployeeNumber                                          */
/*                                                          */
/************************************************************/

#define TV_EmployeeNumber(TM_APPL|TM_PRIM|2)

typedef ASN1INT ASN1T_EmployeeNumber;

class EXTERN ASN1C_EmployeeNumber :
   public ASN1CType
{
protected:
   ASN1T_EmployeeNumber& msgData;
public:
   ASN1C_EmployeeNumber (ASN1T_EmployeeNumber& data);
   ASN1C_EmployeeNumber (
      ASN1MessageBufferIF& msgBuf, ASN1T_EmployeeNumber& data);

   // standard encode/decode methods (defined in ASN1CType base class):
   // int Encode ();
   // int Decode ();

   // stream encode/decode methods:
   int EncodeTo (ASN1MessageBufferIF& msgBuf);
   int DecodeFrom (ASN1MessageBufferIF& msgBuf);
} ;

EXTERN int asn1E_EmployeeNumber (ASN1CTXT* pctxt,
   ASN1T_EmployeeNumber *pvalue, ASN1TagType tagging);

EXTERN int asn1D_EmployeeNumber (ASN1CTXT* pctxt,
   ASN1T_EmployeeNumber *pvalue, ASN1TagType tagging, int length);
```

Note the two main differences between this and the C version:

1. The use of the 'ASN1T_' prefix on the type definition. The C++ version uses the 'ASN1T_' prefix for the typedef and the 'ASN1C_' prefix for the control class definition.

2. The inclusion of the 'ASN1C_EmployeeNumber' control class.

As of ASN1C version 5.6, control classes are not automatically generated for all ASN.1 types. The only types they are generated for are those determined to be *Protocol Data Units* (of PDU's for short). A PDU is a top-level message type in a specification. These are the only types control classes are required for because the only purpose of a control class is to provide the user with a simplified calling interface for encoding and decoding a message. They are not used in any of the ASN1C internally generated logic (the exception to this rule is the XER / XML encoding rules where they are used internally and still must be generated for all types).

A type is determined to be a PDU in two different ways:

1. If it is explicitly declared to be PDU via the <isPDU/> configuration setting or -pdu command-line option.

2. If no explicit declarations exist, a type is determined to be a PDU if it is not referenced by any other types.

In the employee sample program, *EmployeeNumber* would not be considered to be a PDU because it is referenced as an element within the *Employee* production. For the purpose of this discussion, we will assume *EmployeeNumber* was explicitly declared to be PDU via a configuration setting or command-line specification.

ASN1C_EmployeeNumber is the control class declaration. The purpose of the control class is to provide a linkage between the message buffer object and the ASN.1 typed object containing the message data. The class provides methods such as *EncodeTo* and *DecodeFrom* for encoding and decoding the contents to the linked objects. It also provides other utility methods to make populating the typed variable object easier.

ASN1C always adds an *ASN1C_* prefix to the production name to form the class name. Most generated classes are derived from the standard *ASN1CType* base class defined in asn1Message.h. The following ASN.1 types cause code to be generated from different base classes:

- BIT STRING – The generated control class is derived from the *ASN1CBitStr* class

- SEQUENCE OF or SET OF with linked list storage – The generated control class is derived from the *ASN1CSeqOfList* base class.

- Defined Type – The generated control class for defined types is derived from the generated base class for the reference type. For example, if we have A ::= INTEGER and B ::= A, then B is a defined type and would inherit from the base class generated for A (class ASN1C_B : public ASN1C_A { …).

These intermediate classes are also derived from the *ASN1CType* base class. Their purpose is the addition of functionality specific to the given ASN.1 type. For example, the *ASN1CBitStr* control class provides methods for setting, clearing and testing bits in the referenced bit string variable.

In the generated control class, the *msgData* member variable is a reference to a variable of the generated type. The constructor takes two arguments – an *Asn1MessageBufferIF* (message buffer interface) object reference and a reference to a variable of the data type to be encoded or decoded. The message buffer object is a work buffer object for encoding or decoding. The interface reference can also be used to specify a stream. Stream classes are derived from this same base class. The data type reference is a reference to the 'ASN1T_' variable that was generated for the data type.

EncodeFrom and *DecodeTo* methods are declared that wrap the respective compiler generated C encode and decode stream functions. Standard *Encode* and *Decode* methods exist in the ASN1CType base class for direct encoding and decoding to a memory buffer. Command-line options may cause additional methods to be generated. For example, if the –print command line argument was specified; a *Print* method is generated to wrap the corresponding C print function.

Specification of the XML encoding rules option (-xer) causes a number of additional methods to be generated for constructed types. These additional methods are implementations of the standard Simple API for XML (SAX) content handling interface used to parse content from XML messages. The *startElement*, *characters*, and *endElement* methods are implemented as well as additional support methods. The control class is also defined to inherit from the *ASN1XERSAXHandler* base class as well as *ASN1CType* (or one of its descendents).

The equivalent C and C++ type definitions for each of the various ASN.1 types follow.

BOOLEAN

The ASN.1 BOOLEAN type is converted into a C type named "ASN1BOOL". In the global include file "asn1type.h", ASN1BOOL is defined to be an "unsigned character".

ASN.1 production: `<name> ::= BOOLEAN`

Generated C code: `typedef ASN1BOOL <name>;`

Generated C++ code: `typedef ASN1BOOL ASN1T_<name>;`

For example, if "`B ::= [PRIVATE 10] BOOLEAN`" was defined as an ASN.1 production, the generated C type definition would be "`typedef ASN1BOOL B`". Note that the tag information is not represented in the type definition. It is handled within the generated encode/decode functions.

The only difference between the C and C++ mapping is the addition of the 'ASN1T_' prefix on the C++ type.

INTEGER

The ASN.1 INTEGER type is converted into one of several different C types depending on constraints specified on the type. By default, an INTEGER with no constraints results in the generation of an "ASN1INT" type. In the global include file "asn1type.h", ASN1INT is defined to be an "int" which is normally a signed 32-bit integer value on most computer systems.

ASN.1 production: `<name> ::= INTEGER`

Generated C code: `typedef ASN1INT <name>;`

Generated C++ code: `typedef ASN1INT ASN1T_<name>;`

Value range constraints can be used to alter the C type used to represent a given integer value. For example, the following declaration from the SNMP SMI specification would cause an ASN1UINT type (mapped to a C unsigned int) to be used:

```
Counter ::= [APPLICATION 1] IMPLICIT INTEGER (0..4294967295)
```

In this case, an ASN1INT could not be used because all values within the given range could not be represented. Other value ranges would cause different integer types to be used that provide the most efficient amount of storage. The following table shows the types that would be used for the different range values:

Min Lower Bound	Max Upper Bound	ASN1C Type	C Type
-128	127	ASN1CHAR	char (signed 8-bit int)
0	255	ASN1OCTET	unsigned char (unsigned 8-bit number)
-32768	32767	ASN1SINT	short (signed 16-bit int)
0	65535	ASN1USINT	unsigned short (unsigned 16-bit int)

-2147483648	2147483647	ASN1INT	int (signed 32-bit integer)
0	0xffffffff	ASN1UINT	unsigned int (unsigned 32-bit integer)

The C type that is used to represent a given integer value can also be altered using the "<ctype>" configuration variable setting. This allows any of the integer types above to be used for a given integer type as well as a 64-bit integer type. The values that can be used with <ctype> are: byte, int16, uint16, int32, uint32, and int64. An example of using this setting is as follows:

Suppose you have the following integer declaration in your ASN.1 source file:

```
MyIntType ::= [APPLICATION 1] INTEGER
```

You could then have ASN1C use a 64-bit integer type for this integer by adding the following declaration to a configuration file to be associated with this module:

```
<production>
   <name>MyIntType</name>
   <ctype>int64</ctype>
</production>
```

The "<intCType>" setting is also available at the module level to specify that the given C integer type be used for all unconstrained integers within the module.

Large Integer Support

In C and C++, the maximum size for an integer type is normally 64 bits (or 32 bits on some older platforms). ASN.1 has no such limitation on integer sizes and some applications (security key values for example) demand larger sizes. In order to accommodate these types of applications, the ASN1C compiler allows an integer to be declared a "big integer" via a configuration file variable (the <isBigInteger/> setting is used to do this – see the section describing the configuration file for full details). When the compiler detects this setting, it will declare the integer to be a character string variable instead of a C int or unsigned int type. The character string would then be populated with a character string representation of the value to be encoded. Only hexadecimal string representations of the integer value are supported in this release.

For example, the following INTEGER type might be declared in the ASN.1 source file:

```
SecurityKeyType ::= [APPLICATION 2] INTEGER
```

Then, in a configuration file used with the ASN.1 definition above, the following declaration can be made:

```
<production>
   <name>SecurityKeyType</name>
   <isBigInteger/>
</production>
```

This will cause the compiler to generate the following type declaration:

```
typedef ASN1ConstCharPtr SecurityKeyType
```

The ASN1ConstCharPtr type is declared to be a 'char*' type for C and a 'const char*' type for C++ in the asn1type.h header file. The SecurityKeyType variable can now be populated with a hexadecimal string for encoding such as the following:

```
SecurityKeyType secKey = "0xfd09874da875cc90240087cd12fd";
```

Note that in this definition the '0x' prefix is required to identify the string as containing hexadecimal characters.

On the decode side, the decoder will populate the variable with the same type of character string after decoding.

There are also a number of run-time functions available for big integer support. This set of functions provides an arbitrary length integer math package that can be used to perform mathematical operations as well as convert values into various string forms. See the *ASN1C C/C++ Common Run-time User's Manual* for a description of these functions.

BIT STRING

The ASN.1 BIT STRING type is converted into a C or C++ structured type containing an integer to hold the number of bits and an array of unsigned characters ("OCTETs") to hold the bit string contents. The number of bits integer specifies the actual number of bits used in the bit string and takes into account any unused bits in the last byte.

The type definition of the contents field depends on how the bit string is specified in the ASN.1 definition. If a size constraint is used, a static array is generated; otherwise, a pointer variable is generated to hold a dynamically allocated string. The decoder will automatically allocate memory to hold a parsed string based on the received length of the string.

In the static case, the length of the character array is determined by adjusting the given size value (which represents the number of bits) into the number of bytes required to hold the bits.

Dynamic BIT STRING

ASN.1 production: `<name> ::= BIT STRING`

Generated C code: `typedef ASN1DynBitStr <name>;`

Generated C++ code: `typedef ASN1TDynBitStr ASN1T_<name>;`

In this case, different base types are used for C and C++. The difference between the two is the C++ version includes constructors that make setting the value a bit easier.

The ASN1DynBitStr type (i.e., the type used in the C mapping) is defined in the asn1type.h header file as follows:

```
typedef struct ASN1DynBitStr {
   ASN1UINT numbits;
   ASN1OCTET* data;
} ASN1DynBitStr;
```

The ASN1TDynBitStr type is defined in the asn1CppTypes.h header file as follows:

```
struct ASN1TDynBitStr : public ASN1DynBitStr {
   // ctors
   ASN1TDynBitStr () : numbits(0) {}
   ASN1TDynBitStr (ASN1UINT _numbits, ASN1OCTET* _data);
   ASN1TDynBitStr (ASN1DynBitStr& _bs);
} ASN1TDynBitStr;
```

Static (sized) BIT STRING

ASN.1 production: `<name> ::= BIT STRING (SIZE (<len>))`

Generated C code:

```
typedef struct {
   int        numbits;
   ASN1OCTET  data[<adjusted_len>*];
} <name>;
```

Generated C++ code:

```
typedef struct {
   int        numbits;
   ASN1OCTET  data[<adjusted_len>*];
   // ctors
   ASN1T_<name> ();
   ASN1T_<name> (ASN1UINT _numbits,
   ASN1OCTET* _data);
} ASN1T_<name>;
```

`* <adjusted_len> = ((<len> - 1)/8) + 1`

For example, the following ASN.1 production:

```
BS ::= [PRIVATE 220] BIT STRING (SIZE (18))
```

Would translate to the following C typedef:

```
typedef struct ASN1T_BS {
   ASN1UINT numbits;
   ASN1OCTET data[3];
} ASN1T_BS;
```

In this case, three octets would be required to hold the 18 bits: eight in the first two bytes, and two in the third.

Note that for C++, ASN1C generates special constructors and assignment operators to make populating a structure easier. In this case, two constructors were generated: a default constructor and one that takes *numbits* and *data* as arguments.

Named Bits

In the ASN.1 standard, it is possible to define an enumerated bit string that specifies named constants for different bit positions. ASN1C provides support for this type of construct by generating symbolic constants and optional macros that can be used to set, clear, or test these named bits. These symbolic constants equate the bit name to the bit number defined in the specification. They can be used with the *rtBitSet, rtBitClear,* and *rtBitTest* run-time functions to set, clear, and test the named bits. In addition, generated C++ code contains an enumerated constant added to the control class with an entry for each of the bit numbers. These entries can be used in calls to the methods of the *ASN1CBitStr* class to set, clear, and test bits.

The *–genBitMacros* command line option can be used to generate macros to set, clear, or test the named bit in a bit string structure. These macros offer better performance then using the run-time functions because all calculations of mask and index values are done at compile time. The drawback is they can result in a large amount of additional generated code.

For example, the following ASN.1 production:

```
NamedBS ::= BIT STRING { bitOne(1), bitTen(10) }
```

Would translate to the following if *–genBitMacros* was specified:

```
/* Named bit constants */

#define NamedBS_bitOne      1

#define SET_BS3_bitOne(bs) \
<code to set bit..>

#define CLEAR_BS3_bitOne(bs) \
<code to clear bit..>

#define TEST_BS3_bitOne(bs) \
<code to test bit..>

#define NamedBS_bitTen      10

#define SET_BS3_bitTen(bs) \
<code to set bit..>

#define CLEAR_BS3_bitTen(bs) \
<code to clear bit..>

#define TEST_BS3_bitTen(bs) \
<code to test bit..>

/* Type definitions */

typedef struct ASN1T_NamedBS {
   ASN1UINT numbits;
   ASN1OCTET data[2];
} NamedBS;
```

The named bit constants would be used to access the data array within the ASN1T_NamedBS type. If bit macros were not generated, the *rtSetBit* function could be used to set the named bit 'bitOne' with the following code:

```
NamedBS bs;
memset (&bs, 0, sizeof(bs));
rtSetBit (bs.data, 10, NamedBS_bitOne);
```

The statement to clear the bit using *rtClearBit* would be as follows:

```
rtClearBit (bs.data, 10, NamedBS_bitOne);
```

Finally, the bit could be tested using *rtTestBit* with the following statement:

```
if (rtTestBit (bs.data, 10, NamedBS_bitOne) {
   ... bit is set
}
```

Note that the compiler generated a fixed length data array for this specification. It did this because the maximum size of the string is known due to the named bits – it must only be large enough to hold the maximum valued named bit constant.

ASN1CBitStr Control Class

When C++ code generation is specified, a control class is generated for operating on the target bit string. This class is derived from the *ASN1CBitStr* class. This class contains methods for operating on bits within the string.

Objects of this class can also be declared inline to make operating on bits within other ASN.1 constructs easier. For example, in a SEQUENCE containing a bit string element the generated type will contain a public member variable containing the 'ASN1T' type that holds the message data. If one wanted to operate on the bit string contained within that element, they could do so by using the *ASN1CBitStr* class inline as follows:

```
ASN1CBitStr bs (<seqVar>.<element>);
bs.set (0);
```

In this example, <seqVar> would represent a generated SEQUENCE variable type and <element> would represent a bit string element within this type.

See the section on the *ASN1CBitStr* class in the *ASN1C C/C++ Common Run-time User's Manual* for details on all of the methods available in this class.

OCTET STRING

The ASN.1 OCTET STRING type is converted into a C structured type containing an integer to hold the number of octets and an array of unsigned characters ("OCTETs") to hold the octet string contents. The number of octets integer specifies the actual number of octets in the contents field.

The allocation for the contents field depends on how the octet string is specified in the ASN.1 definition. If a size constraint is used, a static array of that size is generated; otherwise, a pointer variable is generated to hold a dynamically allocated string. The decoder will automatically allocate memory to hold a parsed string based on the received length of the string.

For C++, constructors and assignment operators are generated to make assigning variables to the structures easier. In addition to the default constructor, a constructor is provided for string or binary data. An assignment operator is generated for direct assignment of a null-terminated string to the structure (note: this assignment operator copies the null terminator at the end of the string to the data).

Dynamic OCTET STRING

ASN.1 production: `<name> ::= OCTET STRING`

Generated C code: `typedef ASN1DynOctStr <name>;`

Generated C++ code: `typedef ASN1TDynOctStr ASN1T_<name>;`

In this case, different base types are used for C and C++. The difference between the two is the C++ version includes constructors, assignment operators, and other helper methods that make it easier to manipulate binary data.

The *ASN1DynOctStr* type (i.e., the type used in the C mapping) is defined in the asn1type.h header file as follows:

```
typedef struct ASN1DynOctStr {
   ASN1UINT numocts;
   ASN1OCTET* data;
} ASN1DynOctStr;
```

The *ASN1TDynOctStr* type is defined in the *ASN1TOctStr.h* header file. This class extends the C *ASN1DynOctStr* class and adds many additional constructors and methods. See the *C/C++ Common Run-time Reference Manual* for a complete description of this class.

Static (sized) OCTET STRING

ASN.1 production: `<name> ::= OCTET STRING (SIZE (<len>))`

Generated C code:

```
typedef struct {
   ASN1UINT   numocts;
   ASN1OCTET  data[<len>];
} <name>;
```

Generated C++ code:

```
typedef struct {
   ASN1UINT   numocts;
   ASN1OCTET  data[<len>];

   // ctors
   ASN1T_<name> ();
   ASN1T_<name> (ASN1UINT _numocts, ASN1OCTET* _data);
   ASN1T_<name> (char* cstring);

   // assignment operators
   ASN1T_<name>& operator= (char* cstring);
} ASN1T_<name>;
```

ENUMERATED

The ASN.1 ENUMERATED type is converted into different types depending on whether C or C++ code is being generated. The C mapping is either a C enum or integer type depending on whether or not the ASN.1 type is extensible or not. The C++ mapping adds a struct wrapper around this type to provide a namespace to aid in making the enumerated values unique across all modules.

C Mapping

ASN.1 production:
```
<name> ::= ENUMERATED (<id1>(<val1>),
                       <id2>(<val2>), ...)
```

Generated code:
```
typedef enum {
   id1 = val1,
   id2 = val2,
   ...
} <name>
```

The compiler will automatically generate a new identifier value if it detects a duplicate within the source specification. The format of this generated identifier is ‘id_n’ where id is the original identifier and n is a sequential number. The compiler will output an informational message when this is done.

A configuration setting is also available to further disambiguate duplicate enumerated item names. This is the “enum prefix” setting that is available at both the module and production levels. For example, the following would cause the prefix “h225” to be added to all enumerated identifiers within the H225 module:

```
<module>
   <name>H225</name>
   <enumPrefix>h225</enumPrefix>
</module>
```

C++ Mapping

ASN.1 production:
```
<name> ::= ENUMERATED (<id1>(<val1>),
                       <id2>(<val2>), ...)
```

Generated code:
```
struct <name> {
   enum Root {
       id1 = val1,
       id2 = val2,
       ...
   }
   [ enum Ext {
       extid1 = extval1,
       ...
   } ]
} ;

typedef ASN1UINT ASN1T_<name>
```

The struct type provides a namespace for the enumerated elements. This allows the same enumerated constant names to be used in different productions within the ASN.1 specification. An enumerated item is specified in the code using the <name>::<id> form.

Every generated definition contains a 'Root' enumerated specification and, optionally, an 'Ext' specification. The 'Root' specification contains the root elements of the type (or all of the elements if it is not an extended type), and the 'Ext' specification contains the extension enumerated items.

The form of the typedef following the struct specification depends on whether or not the enumerated type contains an extension marker or not. If a marker is present, it means the type can contain values outside the root enumeration. An ASN1UINT is always used in the final typedef to ensure a consistent size of an enumerated variable and to handle the case of unknown extension values.

NULL

The ASN.1 NULL type does not generate an associated C or C++ type definition.

OBJECT IDENTIFIER

The ASN.1 OBJECT IDENTIFIER type is converted into a C or C++ structured type to hold the subidentifier values that make up the object identifier.

ASN.1 production: `<name> ::= OBJECT IDENTIFIER`

Generated C code: `typedef ASN1OBJID <name>;`

Generated C++ code: `typedef ASN1TObjId ASN1T_<name>;`

In this case, different base types are used for C and C++. The difference between the two is the C++ version includes constructors and assignment operators that make setting the value a bit easier.

The ASN1OBJID type (i.e., the type used in the C mapping) is defined in asn1type.h to be the following:

```
typedef struct {
   ASN1UINT numids;  /* number of subidentifiers */
   ASN1UINT subid[ASN_K_MAXSUBIDS];/* subidentifier values */
} ASN1OBJID;
```

The constant "ASN_K_MAXSUBIDS" specifies the maximum number of sub-identifiers that can be assigned to a value of the type. This constant is set to 128 as per the ASN.1 standard. The value of this constant can be changed to a lower number for applications with restricted memory requirements.

The *ASN1TObjId* type used in the C++ mapping is defined in *ASN1TObjId.h*. This class extends the C *ASN1OBJID* structure and adds many additional constructors and helper methods. See the *ASN1C C/C++ Common Run-time Reference Manual* for more details.

RELATIVE-OID

The ASN.1 RELATIVE-OID type is converted into a C or C++ structured type that is identical to that of the OBJECT IDENTIFIER described above:

ASN.1 production: `<name> ::= RELATIVE-OID`

Generated C code: `typedef ASN1OBJID <name>;`

Generated C++ code: `typedef ASN1TObjId ASN1T_<name>;`

A RELATIVE-OID is identical to an OBJECT IDENTIFIER except that it does not contain the restriction on the initial two arc values that they fall within a certain range (see the X.680 standard for more details on this).

REAL

The ASN.1 REAL type is mapped to the C type "ASN1REAL". In the global include file "asn1type.h", ASN1REAL is defined to be a "double".

ASN.1 production: `<name> ::= REAL`

Generated C code: `typedef ASN1REAL <name>;`

Generated C++ code: `typedef ASN1REAL ASN1T_<name>;`

SEQUENCE

This section discusses the mapping of an ASN.1 SEQUENCE type to C. The C++ mapping is similar but there are some differences. These are discusses in the *C++ Mapping of SEQUENCE* subsection at the end of this section.

An ASN.1 SEQUENCE is a constructed type consisting of a series of element definitions. These elements can be of any ASN.1 type including other constructed types. For example, it is possible to nest a SEQUENCE definition within another SEQUENCE definition as follows:

```
A ::= SEQUENCE {
   x  SEQUENCE {
      a1 INTEGER,
      a2 BOOLEAN
   },
   y OCTET STRING SIZE (10)
}
```

In this example, the production has two elements – x and y. The nested SEQUENCE x has two additional elements – a1 and a2.

The ASN1C compiler first recursively pulls all of the embedded constructed elements out of the SEQUENCE and forms new temporary types. The names of the temporary types are of the form <name>_<element-name1>_<element-name2>_ … <element-nameN>. For example, in the definition above, two temporary types would be generated: A_x and A_y (A_y is generated because a static OCTET STRING maps to a C++ struct type).

The general form is as follows:

ASN.1 production:

```
<name> ::= SEQUENCE {
    <element1-name> <element1-type>,
    <element2-name> <element2-type>,
    ...
}
```

Generated C code:

```
typedef struct {
   <type1> <element1-name>;
   <type2> <element2-name>;
   ...
} <name>;
```

- or -

```
typedef struct {
   ...
} <tempName1>

typedef struct {
   ...
} <tempName2>

typedef struct {
   <tempName1> <element1-name>;
   <tempName2> <element2-name>;
   ...
} <name>;
```

The <type1> and <type2> placeholders represent the equivalent C types for the ASN.1 types <element1-type> and <element2-type> respectively. This form of the structure will be generated if the internal types are primitive. <tempName1> and <tempName2> are formed using the algorithm described above for pulling structured types out of the definition. This form is used for constructed elements and elements that map to structured C types.

The example above would result in the following generated C typedefs:

```
typedef struct _A_x {
   ASN1INT   a1;
   ASN1BOOL  a2;
} A_x;

typedef struct A_y {
   ASN1UINT numocts;
   ASN1OCTET data[10];
} A_y;

typedef struct _A {
   A_x  x;
   A_y  y;
} A;
```

In this case, elements x and y map to structured C types, so temporary typedefs are generated.

In the case of nesting levels greater than two, all of the intermediate element names are used to form the final name. For example, consider the following type definition that contains three nesting levels:

```
X ::= SEQUENCE {
   a  SEQUENCE {
      aa  SEQUENCE { x INTEGER, y BOOLEAN },
      bb  INTEGER
   }
}
```

In this case, the generation of temporary types results in the following equivalent type definitions:

```
X-a-aa ::= SEQUENCE { x INTEGER, y BOOLEAN }

X-a ::= SEQUENCE { aa X-a-aa, bb INTEGER }

X ::= SEQUENCE { X-a a }
```

Note that the name for the aa element type is X-a-aa. It contains both the name for a (at level 1) and aa (at level 2). This is a change from v5.1x and lower where only that production name and last element name would be used (i.e., X-aa). The change was made to ensure uniqueness of the generated names when multiple nesting levels are used.

Note that although the compiler can handle embedded constructed types within productions, it is generally not considered good style to define productions this way. It is much better to manually define the constructed types for use in the final production definition. For example, the production defined at the start of this section can be rewritten as the following set of productions:

```
X ::= SEQUENCE {
   a1 INTEGER,
   a2 BOOLEAN
}

Y ::= OCTET STRING
```

```
A ::= SEQUENCE {
   X x,
   Y y
}
```

This makes the generated code easier to understand for the end user.

Unnamed Elements

Note: as of X.680, unnamed elements are not allowed – elements must be named. ASN1C still provides backward compatibility support for this syntax however.

In an ASN.1 SEQUENCE definition, the <element-name> tokens at the beginning of element declarations are optional. It is possible to include only a type name without a field identifier to define an element. This is normally done with defined type elements, but can be done with built-in types as well. An example of a SEQUENCE with unnamed elements would be as follows:

```
AnInt ::= [PRIVATE 1] INTEGER

Aseq ::= [PRIVATE 2] SEQUENCE {
   x       INTEGER,
           AnInt
}
```

In this case, the first element (x) is named and the second element is unnamed.

ASN1C handles this by generating an element name using the type name with the first character set to lower case. For built-in types, a constant element name is used for each type (for example, aInt is used for INTEGER). There is one caveat, however. ASN1C cannot handle multiple unnamed elements in a SEQUENCE or SET with the same type names. Element names must be used in this case to distinguish the elements.

So, for the example above, the generated code would be as follows:

```
typedef ASN1INT AnInt;

typedef struct Aseq {
   ASN1INT   x;
   AnInt     anInt;
} Aseq;
```

OPTIONAL keyword

Elements within a sequence can be declared to be optional using the OPTIONAL keyword. This indicates that the element is not required in the encoded message. An additional construct is added to the generated code to indicate whether an optional element is present in the message or not. This construct is a bit structure placed at the beginning of the generated sequence structure. This structure always has variable name ‘m’ and contains single-bit elements of the form ‘<element-name>Present’ as follows:

```
struct {
   unsigned <element-name1>Present : 1,
   unsigned <element-name2>Present : 1,
   ...
} m;
```

In this case, the elements included in this construct correspond to only those elements marked as OPTIONAL within the production. If a production contains no optional elements, the entire construct is omitted.

For example, we will change the production in the previous example to make both elements optional:

```
Aseq ::= [PRIVATE 2] SEQUENCE {
   x       INTEGER OPTIONAL,
           AnInt OPTIONAL
}
```

In this case, the following C typedef is generated:

```
typedef struct Aseq {
   struct {
      unsigned xPresent : 1,
      unsigned anIntPresent : 1
   } m;
   ASN1INT   x;
   AnInt     anInt;
} Aseq;
```

When this structure is populated for encoding, the developer must set the xPresent and anIntPresent flags accordingly to indicate whether the elements are to be included in the encoded message or not. Conversely, when a message is decoded into this structure, the developer must test the flags to determine if the element was provided in the message or not.

The C++ version of the compiler will generate a constructor for the structured type for a SEQUENCE if OPTIONAL elements are present. This constructor will set all optional bits to zero when a variable of the structured type is declared. The programmer therefore does not have to be worried about clearing bits for elements that are not used; only with setting bits for the elements that are to be encoded.

DEFAULT keyword

The DEFAULT keyword allows a default value to be specified for elements within the SEQUENCE. ASN1C will parse this specification and treat it as it does an optional element. Note that the value specification is only parsed in simple cases for primitive values. It is up to the programmer to provide the value in complex cases. For BER encoding, a value must be specified be it the default or other value.

For DER or PER, it is a requirement that no value be present in the encoding for the default value. For integer and boolean default values, the compiler automatically generates code to handle this requirement based on the value in the structure. For other values, an optional present flag bit is generated. The programmer must set this bit to false on the encode side to specify default value selected. If this is done, a value is not encoded into the message. On the decode side, the developer must test for present bit not set. If this is the case, the default value specified in the ASN.1 specification must be used and the value in the structure ignored.

Extension Elements

If the SEQUENCE type contains an open extension field (i.e., a … at the end of the specification or a …, … in the middle), a special element will be inserted to capture encoded extension elements for inclusion in the final encoded message. This element will be of type *Asn1RTDList* and have the name *extElem1*. This is a linked list of open type fields. Each entry in the list is of type *ASN1OpenType*. The fields will contain complete encodings of any extension elements that may have been present in a message when it is decoded. On subsequent encode of the type, the extension fields will be copied into the new message.

The *–noOpenExt* command line option can be used to alter this default behavior. If this option is specified, the *extElem1* element is not included in the generated code and extension data that may be present in a decoded message is simply dropped.

If the SEQUENCE type contains an extension marker and extension elements, then the actual extension elements will be present in addition to the *extElem1* element. These elements will be treated as optional elements whether they were declared that way or not. The reason is because a version 1 message could be received that does not contain the elements.

Additional bits will be generated in the bit mask if version brackets are present. These are groupings of extended elements that typically correspond to a particular version of a protocol. An example would be as follows:

```
TestSequence ::= SEQUENCE {
   item-code    INTEGER (0..254),
   item-name    IA5String (SIZE (3..10)) OPTIONAL,
   ... ! 1,
   urgency      ENUMERATED { normal, high } DEFAULT normal,
   [[ alternate-item-code       INTEGER (0..254),
      alternate-item-name       IA5String (SIZE (3..10)) OPTIONAL
   ]]
}
```

In this case, a special bit flag will be added to the mask structure to indicate the presence or absence of the entire element block. This will be of the form "_v#ExtPresent" where # would be replaced by the sequential version number. In the example above, this number would be three (two would be the version extension number of the urgency field). Therefore, the generated bit mask would be as follows:

```
struct {
      unsigned item_namePresent : 1;
      unsigned urgencyPresent : 1;
      unsigned _v3ExtPresent : 1;
      unsigned alternate_item_namePresent : 1;
   } m;
```

In this case, the setting of the _v3ExtPresent flag would indicate the presence or absence of the entire version block. Note that it is also possible to have optional items within the block (alternate-item-name).

C++ Mapping of SEQUENCE

As mentioned earlier, the C++ mapping is very similar to the C mapping. However, there are some important differences:

1. As with all C++ types, the prefix *ASN1T_* is added before the typename to distinguish the data class from the control class (the control class contains an *ASN1C_* prefix).

2. A default constructor is generated to initialize the structure elements. This constructor will zero out all elements and set any simple default values that may have been specified in the ASN.1 definition.

3. If the -genCopy command line switch was specified, a copy constructor will be generated to allow an instance of the data contained within a PDU control class object to be copied.

4. Also if -genCopy was specified, a destructor is generated if the type contains dynamic fields. This destructor will free all memory held by the type when the object is deleted or goes out of scope.

SET

The ASN.1 SET type is converted into a C or C++ structured type that is identical to that for SEQUENCE as described in the previous section. The only difference between SEQUENCE and SET is that elements may be transmitted in any order in a SET whereas they must be in the defined order in a SEQUENCE. The only impact this has on ASN1C is in the generated decoder for a SET type.

The decoder must take into account the possibility of out-of-order elements. This is handled by using a loop to parse each element in the message. Each time an item is parsed, an internal mask bit within the decoder is set to indicate the element was received. The complete set of received elements is then checked after the loop is completed to verify all required elements were received.

SEQUENCE OF

The ASN.1 SEQUENCE OF type is converted into one of the following C/C++ types:

- A doubly-linked list structure (*Asn1RTDList* for C, or *ASN1TSeqOfList*, a class derived from *Asn1RTDList*, for C++)
- A structure containing an integer count of elements and a pointer to hold an array of the referenced data type (a dynamic array)
- A structure containing an integer count of elements and a fixed-sized array of the referenced data type (a static array)

The linked list option is the default for constructed types. An array is used for a sequence of primitive types. The allocation for the contents field of the array depends on how the SEQUENCE OF is specified in the ASN.1 definition. If a size constraint is used, a static array of that size is generated; otherwise, a pointer variable is generated to hold a dynamically allocated array of values. The decoder will automatically allocate memory to hold parsed SEQUENCE OF data values.

The type used for a given SEQUENCE OF construct can be modified by the use of a configuration item. The <storage> qualifier with the 'dynamic' keyword can be used at the global, module, or production level to specify that dynamic memory (i.e., a pointer) is used for the array. The syntax of this qualifier is as follows:

```
<storage>dynamic</storage>
```

The 'list' keyword can also be used in a similar fashion to specify the use of a linked-linked structure to hold the elements:

```
<storage>list</storage>
```

See the section entitled Compiler Configuration File for further details on setting up a configuration file.

Dynamic SEQUENCE OF Type

ASN.1 production:

```
<name> ::= SEQUENCE OF <type>
```

Generated C code:

```
typedef struct {
  int        n;
  <type>*    elem;
} <name>;
```

Generated C++ code:

```
typedef struct {
  int        n;
  <type>*    elem;
} ASN1T_<name>;
```

Note that parsed values can be accessed from the dynamic data variable just as they would be from a static array variable; i.e., an array subscript can be used (ex: elem[0], elem[1]...).

Static (sized) SEQUENCE OF Type

ASN.1 production: `<name> ::= SEQUENCE SIZE <len> OF <type>`

Generated C code:

```
typedef struct {
  int        n;
  <type>     elem[<len>];
} <name>;
```

Generated C++ code:

```
typedef struct {
  int        n;
  <type>     elem[<len>];
} ASN1T_<name>;
```

List-based SEQUENCE OF Type

A doubly-linked list header type (*Asn1RTDList*) is used for the typedef if the list storage configuration setting is used (see above). This can be used for either a sized or unsized SEQUENCE OF construct. The generated C or C++ code is as follows:

Generated C code: `typedef Asn1RTDList <name>;`

Generated C++ code: `typedef ASN1TSeqOfList ASN1T_<name>;`

The type definition of the Asn1RTDList structure can be found in the asn1type.h header file. The common run-time utility functions beginning with the prefix *rtDList* are available for initializing and adding elements to the list. See the Common Run-time Functions section for a full description of these functions.

In addition to the *Asn1RTDList* C structure and C functions, a C++ class is provided for linked list support. This is the *ASN1CSeqOfList* class. This class provides methods for adding and deleting elements to and from lists and an iterator interface for traversing lists. See the *ASN1CSeqOfList* section in the C++ Run-Time Classes area for details on all of the methods available in this class.

Generation of Temporary Types for SEQUENCE OF Elements

As with other constructed types, the <type> variable can reference any ASN.1 type, including other ASN.1 constructed types. Therefore, it is possible to have a SEQUENCE OF SEQUENCE, SEQUENCE OF CHOICE, etc.

When a constructed type or type that maps to a C structured type is referenced, a temporary type is generated for use in the final production. The format of this temporary type name is as follows:

```
<prodName>_element
```

In this definition, <prodName> refers to the name of the production containing the SEQUENCE OF type.

For example, a simple (and very common) single level nested SEQUENCE OF construct might be as follows:

```
A ::= SEQUENCE OF SEQUENCE { INTEGER a, BOOLEAN b }
```

In this case, a temporary type is generated for the element of the SEQUENCE OF construct. This results in the following two equivalent ASN.1 types:

```
A-element ::= SEQUENCE { INTEGER a, BOOLEAN b }

A ::= SEQUENCE OF A-element
```

These types are then converted into the equivalent C or C++ typedefs using the standard mapping that was previously described.

SEQUENCE OF Type Elements in Other Constructed Types

Frequently, a SEQUENCE OF construct is used to define an array of some common type in an element in some other constructed type (for example, a SEQUENCE). An example of this is as follows:

```
SomePDU ::= SEQUENCE {
   addresses SEQUENCE OF AliasAddress,
   ...
}
```

Normally, this would result in the addresses element being pulled out and used to create a temporary type with a name equal to "SomePDU-addresses" as follows:

```
SomePDU-addresses ::= SEQUENCE OF AliasAddress

SomePDU ::= SEQUENCE {
   addresses SomePDU-addresses,
   ...
}
```

However, when the SEQUENCE OF element references a simple defined type as above with no additional tagging or constraint information, an optimization is done to cut down on the size of the generated code. This optimization is to generate a common name for the new temporary type that can be used for other similar references. The form of this common name is as follows:

```
_SeqOf<elementProdName>
```

So instead of this:

```
SomePDU-addresses ::= SEQUENCE OF AliasAddress
```

The following equivalent type would be generated:

```
_SeqOfAliasAddress ::= SEQUENCE OF AliasAddress
```

The advantage is that the new type can now be easily reused if "SEQUENCE OF AliasAddress" is used in any other element declarations. Note the (illegal) use of an underscore in the first position. This is to ensure that no name collisions occur with other ASN.1 productions defined within the specification.

An example of the savings of this optimization can be found in H.225. The above element reference is repeated 25 different times in different places. The result is the generation of one new temporary type that is referenced in 25 different places. Without this optimization, 25 unique types with the same definition would have been generated.

SET OF

The ASN.1 SET OF type is converted into a C or C++ structured type that is identical to that for SEQUENCE OF as described in the previous section.

CHOICE

The ASN.1 CHOICE type is converted into a C or C++ structured type containing an integer for the choice tag value (t) followed by a union (u) of all of the equivalent types that make up the CHOICE elements.

The tag value is simply a sequential number starting at one for each alternative in the CHOICE. A #define constant is generated for each of these values. The format of this constant is "T_<name>_<element-name>" where <name> is the name of the ASN.1 production and <element-name> is the name of the CHOICE alternative. If a CHOICE alternative is not given an explicit name, then <element-name> is automatically generated by taking the type name and making the first letter lowercase (this is the same as was done for the ASN.1 SEQUENCE type with unnamed elements). If the generated name is not unique, a sequential number is appended to make it unique.

The union of choice alternatives is made of the equivalent C or C++ type definition followed by the element name for each of the elements. The rules for element generation are essentially the same as was described for SEQUENCE above. Constructed types or elements that map to C structured types are pulled out and temporary types are created. Unnamed elements names are automatically generated from the type name by making the first character of the name lowercase.

One difference between temporary types used in a SEQUENCE and in a CHOICE is that a pointer variable will be generated for use within the CHOICE union construct.

ASN.1 production:

```
<name> ::= CHOICE {
 <element1-name> <element1-type>,
 <element2-name> <element2-type>,
 ...
}
```

Generated C code:

```
#define T_<name>_<element1-name> 1
#define T_<name>_<element2-name> 2
...

typedef struct {
   int       t;
   union {
      <type1> <element1-name>;
      <type2> <element2-name>;
      ...
   } u;
} <name>;
```

- or -

```
typedef struct {
    ...
} <tempName1>;

typedef struct {
    ...
} <tempName2>;

typedef struct {
    int       t;
    union {
       <tempName1>* <element1-name>;
       <tempName2>* <element2-name>;
       ...
     } u;
} <name>;
```

If the *–static* command line option or <*storage*> *static* <*/storage*> configuration variable is set for the given production, then pointers will not be used for the variable declarations. This is true for the C case only – for C++, pointers must be used due to the fact that the generated code will not compile if constructors are used in a non-pointer variable within a union construct.

The C++ mapping is the same with the exception that the 'ASN1T_' prefix is added to the generated type name.

<type1> and <type2> are the equivalent C types representing the ASN.1 types <element1-type> and <element2-type> respectively. <tempName1> and <tempName2> represent the names of temporary types that may have been generated as the result of using constructed types within the definition.

Choice alternatives may be unnamed, in which case <element-name> is derived from <element-type> by making the first letter lowercase. One needs to be careful when nesting CHOICE structures at different levels within other nested ASN.1 structures (SEQUENCEs, SETs, or other CHOICEs). A problem arises when CHOICE element names at different levels are not unique (this is likely when elements are unnamed). The problem is that generated tag constants are not guaranteed to be unique since only the production and end element names are used.

The compiler gets around this problem by checking for duplicates. If the generated name is not unique, a sequential number is appended to make it unique. The compiler outputs an informational message when it does this.

An example of this can be found in the following production:

```
C ::= CHOICE {
   [0] INTEGER,
   [1] CHOICE {
      [0] INTEGER,
      [1] BOOLEAN
   }
}
```

This will produce the following C code:

```
#define T_C_aInt        1
#define T_C_aChoice     2
#define T_C_aInt_1      1
#define T_C_aBool       2

typedef struct {
   int t;
   union {
      ASN1INT aInt;
      struct {
         int t;
         union {
            ASN1INT aInt;
            ASN1BOOL aBool;
         } u;
      } aChoice;
   } C;
```

Note that '_1' was appended to the second instance of 'T_C_aInt'. Developers must take care to ensure they are using the correct tag constant value when this happens.

Populating Generated Choice Structures

Populating generated CHOICE structures is more complex then for other generated types due to the use of pointers within the union construct. As previously mentioned, the use of pointers with C can be prevented by using the *–static* command line option. If this is done, the elements within the union construct will be standard inline variable declaration and can be populated directly. Otherwise, the methods listed below can be used to populate the variables.

The recommended way to populate the pointer elements is to declare variables of the embedded type to be used on the stack prior to populating the CHOICE structure. The embedded variable would then be populated with the data to be encoded and then the address of this variable would be plugged into the CHOICE union pointer field.

Consider the following definitions:

```
AsciiString  ::= [PRIVATE 28] OCTET STRING
EBCDICString ::= [PRIVATE 29] OCTET STRING
String ::= CHOICE { AsciiString, EBCDICString }
```

This would result in the following type definitions:

```
typedef ASN1DynOctStr AsciiString;
typedef ASN1DynOctStr EBCDICString;

typedef struct String {
   int t;
   union {
      /* t = 1 */
      AsciiString *asciiString;
      /* t = 2 */
      EBCDICString *eBCDICString;
   } u;
} String;
```

To set the *AsciiString* choice value, one would first declare an *AsciiString* variable, populate it, and then plug the address into a variable of type *String* structure as follows:

```
AsciiString asciiString;
String      string;

asciiString = "Hello!";
string.t = T_String_AsciiString;
string.u.asciiString = &asciiString;
```

It is also possible to allocate dynamic memory for the CHOICE union option variable; but one must be careful to release this memory when done with the structure.

Open Type

Note: The X.680 Open Type replaces the X.208 ANY or ANY DEFINED BY constructs. An ANY or ANY DEFINED BY encountered within an ASN.1 module will result in the generation of code corresponding to the Open Type described below.

An *Open Type* as defined in the X.680 standard is specified as a reference to a *Type Field* in an *Information Object Class*. The most common form of this is when the *Type* field in the built-in TYPE-IDENTIFIER class is referenced as follows:

```
      TYPE-IDENTIFIER.&Type
```

See the section in this document on Information Objects for a more detailed explanation.

The *Open Type* is converted into a C or C++ structure used to model a dynamic OCTET STRING type. This structure contains a pointer and length field. The pointer is assumed to point at a string of previously encoded ASN.1 data. When a message containing an open type is decoded, the address of the open type contents field is stored in the pointer field and the length of the component is stored in the length field.

ASN.1 production: `<name> ::= ANY`

Generated C code: `typedef ASN1OpenType <name>;`

Generated C++ code: `typedef ASN1TOpenType <name>;`

The difference between the two types is the C++ version contains constructors to initialize the value to zero or to a given open type value.

If the –tables command line option is selected and the ASN.1 type definition references a table constraint, the code generated is different. In this case, *ASN1OpenType* above is replaced with *ASN1Object* (or *ASN1TObject* for C++). This is defined in *asn1type.h* as follows:

```
typedef struct {  /* generic table constraint value holder */
   ASN1OpenType encoded;
   void*        decoded;
   int          index;      /* table index */
} ASN1Object;
```

This allows a value of any ASN.1 type to be represented in both encoded and decoded forms. Encoded form is the open type form shown above. It is simply a pointer to a byte buffer and a count of the number of byes in the encoded message component. The decoded form is pointer to a variable of a specific type. The pointer is void because there could be a potentially large number of different types that can be represented in the table constraint used to constrain a type field to a given set of values. The *index* member of the type is for internal use by table constraint processing functions to keep track of which row in a table is being referenced.

Character String Types

As of version 5.0 and above, character string types are now built into the compiler. Previous versions used compiled definitions based on the OCTET STRING base type to model these types. All 8-bit character character-string types now are derived from the C character pointer (char*) base type. This pointer is used to hold a null-terminated C string for encoding/decoding. For encoding, the string can either be static (i.e., a string literal or address of a static buffer) or dynamic. The decoder allocates dynamic memory from within its context to hold the memory for the string. This memory is released when the *rtMemFree* function is called.

The useful character string types in ASN.1 are as follows:

```
UTF8String      ::= [UNIVERSAL 12] IMPLICIT OCTET STRING
NumericString   ::= [UNIVERSAL 18] IMPLICIT IA5String
PrintableString ::= [UNIVERSAL 19] IMPLICIT IA5String
T61String       ::= [UNIVERSAL 20] IMPLICIT OCTET STRING
VideotexString  ::= [UNIVERSAL 21] IMPLICIT OCTET STRING
IA5String       ::= [UNIVERSAL 22] IMPLICIT OCTET STRING
UTCTime         ::= [UNIVERSAL 23] IMPLICIT GeneralizedTime
GeneralizedTime ::= [UNIVERSAL 24] IMPLICIT IA5String
GraphicString   ::= [UNIVERSAL 25] IMPLICIT OCTET STRING
VisibleString   ::= [UNIVERSAL 26] IMPLICIT OCTET STRING
GeneralString   ::= [UNIVERSAL 27] IMPLICIT OCTET STRING
UniversalString ::= [UNIVERSAL 28] IMPLICIT OCTET STRING
BMPString       ::= [UNIVERSAL 30] IMPLICIT OCTET STRING
```

```
ObjectDescriptor ::= [UNIVERSAL 7] IMPLICIT GraphicString
```

Of these, all are represented by char* pointers except for the BMPString and UniversalString types. The BMPString is a 16-bit character string for which the following structure is used:

```
typedef struct {
   ASN1UINT       nchars;
   ASN116BITCHAR* data;
} Asn116BitCharString;
```

The ASN116BITCHAR type used in this definition is defined to be an “unsigned short”.

See the *rtBMPToCString*, *rtBMPToNewCString*, and the *rtCToBMPString* run-time function descriptions for information on utilities that can convert standard C strings to and from BMP string format.

The UniversalString type is a 32-bit character string for which the following structure is used:

```
typedef struct {
   ASN1UINT       nchars;
   ASN132BITCHAR* data;
} Asn132BitCharString;
```

The ASN132BITCHAR type used in this definition is defined to be an “unsigned int”.

See the *rtUCSToCString*, *rtUCSToNewCString*, and the *rtCToUCSString* run-time function descriptions for information on utilities that can convert standard C strings to and from Universal Character Set (UCS-4) string format. See also the *rtUCSToWCSString* and *rtWCSToUCSString* for information on utilities that can convert standard wide character string to and from UniversalString type.

Utilities are also provided for working with UTF-8 string dataThe contents for this string type are assumed to contain the UTF-8 encoding of a character string. The UTF-8 encoding for a standard ASCII string is simply the string itself. For Unicode strings represented in C/C++ using the wide character type (wchar_t), the run-time functions *rtUTF8ToWCS* and *rtWCSToUTF8* can be used for converting to and from Unicode. The function *rtValidateUTF8* can be used to ensure that a given UTF-8 encoding is valid. See the *Run-Time Common Library* section for a complete description of these functions.

Time String Types

The ASN.1 *GeneralizedTime* and *UTCTime* types are mapped to standard C/C++ null-terminated character string types.

The C++ version of the product contains additional control classes for parsing and formatting time string values. When C++ code generation is specified, a control class is generated for operating on the target time string. This class is derived from the *ASN1CGeneralizedTime* or *ASN1CUTCTime* class for *GeneralizedTime* or *UTCTime* respectively. These classes contain methods for formatting or parsing time components such as month, day, year etc. from the strings.

Objects of these classes can be declared inline to make the task of formatting or parsing time strings easier. For example, in a SEQUENCE containing a time string element the generated type will contain a public member variable containing the ‘ASN1T’ type that holds the message data. If one wanted to operate on the time string contained within that element, they could do so by using one of the time string classes inline as follows:

```
ASN1CGeneralizedTime gtime (msgbuf, <seqVar>.<element>);
gtime.setMonth (ASN1CTime::November);
```

In this example, <seqVar> would represent a generated SEQUENCE variable type and <element> would represent a time string element within this type.

See the *ASN1CTime, ASN1CGeneralizedTime* and *ASN1CUTCTIME* subsections in the C++ Run-Time Classes section for details on all of the methods available in these classes.

EXTERNAL Type

The ASN.1 EXTERNAL type is a useful type used to include non-ASN.1 or other data within an ASN.1 encoded message. This type is described using the following ASN.1 SEQUENCE:

```
EXTERNAL  ::=  [UNIVERSAL 8] IMPLICIT SEQUENCE {
   direct-reference  OBJECT IDENTIFIER OPTIONAL,
   indirect-reference  INTEGER OPTIONAL,
   data-value-descriptor  ObjectDescriptor  OPTIONAL,
   encoding  CHOICE {
      single-ASN1-type  [0] ANY,
      octet-aligned     [1] IMPLICIT OCTET STRING,
      arbitrary         [2] IMPLICIT BIT STRING
   }
}
```

The ASN.1 compiler is used to create a meta-definition for this structure. This code will be always generated in the *Asn1External.h* and *Asn1External.c/cpp* files. The code will only be generated if the given ASN.1 source specification requires this definition. The resulting C structure is populated just like any other compiler-generated structure for working with ASN.1 data.

> NOTE: It is recommended that if a specification contains multiple ASN.1 source files that reference EXTERNAL, all of these source files be compiled with a single ASN1C call in order to ensure that only a singled copy of the *Asn1External* source files are generated.

EMBEDDED PDV Type

The ASN.1 EMBEDDED PDV type is a useful type used to include non-ASN.1 or other data within an ASN.1 encoded message. This type is described using the following ASN.1 SEQUENCE:

```
EmbeddedPDV ::=  [UNIVERSAL 11] IMPLICIT SEQUENCE {
   identification CHOICE {
      syntaxes SEQUENCE {
         abstract OBJECT IDENTIFIER,
         transfer OBJECT IDENTIFIER
      },
      syntax OBJECT IDENTIFIER,
      presentation-context-id INTEGER,
      context-negotiation SEQUENCE {
         presentation-context-id INTEGER,
```

```
         transfer-syntax OBJECT IDENTIFIER
      },
      transfer-syntax OBJECT IDENTIFIER,
      fixed NULL
   },
   data-value-descriptor ObjectDescriptor OPTIONAL,
   data-value OCTET STRING
}( WITH COMPONENTS { ... , data-value-descriptor ABSENT})
```

The ASN.1 compiler is used to create a meta-definition for this structure. This code will be always generated in the *Asn1EmbeddedPDV.h* and *Asn1EmbeddedPDV.c/cpp* files. The code will only be generated if the given ASN.1 source specification requires this definition. The resulting C structure is populated just like any other compiler-generated structure for working with ASN.1 data.

> NOTE: It is recommended that if a specification contains multiple ASN.1 source files that reference EXTERNAL, all of these source files be compiled with a single ASN1C call in order to ensure that only a singled copy of the *Asn1External* source files are generated.

Parameterized Types

The compiler can parse parameterized type definitions and references as specified in the X.683 standard. These types allow dummy parameters to be declared that will be replaced with actual parameters when the type is referenced. This is similar to templates in C++.

A simple and common example of the use of parameterized types is for the declaration of an upper bound on a sized type as follows:

```
SizedOctetString{INTEGER:ub} ::= OCTET STRING (SIZE (1..ub))
```

In this definition, 'ub' would be replaced with an actual value when the type is referenced. For example, a sized octet string with an upper bound of 32 would be declared as follows:

```
OctetString32 ::= SizedOctetString{32}
```

The compiler would handle this in the same way as if the original type was declared to be an octet string of size 1 to 32. That is, it will generate a C structure containing a static byte array of size 32 as follows:

```
typedef struct  OctetString32 {
   ASN1UINT     numocts;
   ASN1OCTET    data[32];
} OctetString32;
```

Another common example of parameterization is the substitution of a given type inside a common container type. For example, security specifications frequently contain a 'signed' parameterized type that allows a digital signature to be applied to other types. An example of this would be as follows:

```
SIGNED { ToBeSigned } ::= SEQUENCE {
         toBeSigned    ToBeSigned,
         algorithmOID  OBJECT IDENTIFIER,
```

```
         paramS        Params,
         signature     BIT STRING
}
```

An example of a reference to this definition would be as follows:

```
      SignedName ::= SIGNED { Name }
```

where ‘Name’ would be another type defined elsewhere within the module.

The compiler performs the substitution to create the proper C typedef for SignedName:

```
typedef struct SignedName {
   Name           toBeSigned;
   ASN1OBJID      algorithmOID;
   Params         paramS;
   ASN1DynBitStr  signature;
} SignedName;
```

When processing parameterized type definitions, the compiler will first look to see if the parameters are actually used in the final generated code. If not, they will simply be discarded and the parameterized type converted to a normal type reference. For example, when used with information objects, parameterized types are frequently used to pass information object set definitions to impose table constraints on the final type. Since table constraints do not affect the code that is generated by the compiler, the parameterized type definition is reduced to a normal type definition and references to it are handled in the same way as defined type references. This can lead to a significant reduction in generated code in cases where a parameterized type is referenced over and over again.

For example, consider the following often-repeated pattern from the UMTS 3GPP specs:

```
ProtocolIE-Field {RANAP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
   id             RANAP-PROTOCOL-IES.&id          ({IEsSetParam}),
   criticality    RANAP-PROTOCOL-IES.&criticality ({IEsSetParam}{@id}),
   value          RANAP-PROTOCOL-IES.&Value       ({IEsSetParam}{@id})
}
```

In this case, IEsSetParam refers to an information object set specification that constrains the values that are allowed to be passed for any given instance of a type referencing a ProtocolIE-Field. The compiler does not add any extra code to check for these values, so the parameter can be discarded (note that this is not true if the –tables compiler option is specified). After processing the Information Object Class references within the construct (refer to the section on “Information Objects” for information on how this is done), the reduced definition for ProtocolIE-Field becomes the following:

```
      ProtocolIE-Field ::= SEQUENCE {
         id            ProtocolIE-ID,
         criticality   Criticality,
         value         ASN.1 OPEN TYPE
      }
```

References to the field are simply replaced with a reference to the ProtocolID-Field typedef.

If –tables is specified, the parameters are used and a new type instance is created in accordance with the rules above.

Value Specifications

The compiler can parse any type of ASN.1 value specification, but the standard version of the ASN1C compiler will only generate code for following value specifications:

- BOOLEAN
- INTEGER
- REAL
- ENUMERATED
- Binary String
- Hexadecimal String
- Character String
- OBJECT IDENTIFER

All value types except INTEGER and REAL cause an "extern" statement to be generated in the header file and a global value assignment to be added to the C or C++ source file. INTEGER and REAL value specifications cause #define statements to be generated.

BOOLEAN Value Specification

A BOOLEAN value causes an "extern" statement to be generated in the header file and a global declaration of type ASN1BOOL to be generated in the C or C++ source file. The mapping of ASN.1 declaration to global C or C++ value declaration is as follows:

ASN.1 production: `<name> BOOLEAN ::= <value>`

Generated code: `ASN1BOOL <name> = <value>;`

INTEGER Value Specification

The INTEGER type causes a #define statement to be generated in the header file of the form 'ASN1V_<valueName>' where <valueName> would be replaced with the name in the ASN.1 source file. The reason for doing this is the common use of INTEGER values for size and value range constraints in the ASN.1 specifications. By generating #define statements, the symbolic names can be included in the source code making it easier to adjust the boundary values on the fly.

For example, the following declaration:

```
ivalue INTEGER ::= 5
```

will cause the following statement to be added to the generated header file:

```
#define ASN1V_ivalue 5
```

The reason the ASN1V_ prefix is added is to prevent collisions with INTEGER value declarations and other declarations such as enumeration items with the same name.

REAL Value Specification

The REAL type causes a #define statement to be generated in the header file of the form 'ASN1V_<valueName>' where <valueName> would be replaced with the name in the ASN.1 source file. By generating #define statements, the symbolic names can be included in the source code making it easier to adjust the boundary values on the fly.

For example, the following declaration:

```
rvalue REAL ::= 5.5
```

will cause the following statement to be added to the generated header file:

```
#define ASN1V_rvalue 5.5
```

The reason the ASN1V_ prefix is added is to prevent collisions with other declarations such as enumeration items with the same name.

Enumerated Value Specification

The mapping of ASN.1 declaration to global C or C++ value declaration is as follows:

ASN.1 production: `<name> <EnumType> ::= <value>`

Generated code: `ASN1UINT <name> = <value>;`

Binary and Hexadecimal String Value Specification

These value specifications cause two global C variables to be generated: a 'numocts' variable describing the length of the string and a 'data' variable describing the string contents. The mapping for a binary string is as follows (note: BIT STRING can also be used as the type in this type of declaration):

ASN.1 production: `<name> OCTET STRING ::= '<bstring>'B`

Generated code:
```
ASN1UINT  <name>_numocts = <length>;
     ASN1OCTET <name>_data[]  = <data>;
```

Hexadecimal string would be the same except the ASN.1 constant would end in a 'H'.

Character String Value Specification

A character string declaration would cause a C or C++ char* declaration to be generated:

ASN.1 production: `<name> <string-type> ::= <value>`

Generated code: `ASN1ConstCharPtr <name> = <value>;`

In this definition, <string-type> could be any of the standard 8-bit characters string types such as IA5String, PrintableString, etc. (note: this version of the compiler does not contain support for value declarations of larger character string type such as BMPString). The ASN1ConstCharPtr type used in the generated code is a type defined in asn1type.h designed to be a char* type for C or const char* type for C++.

Object Identifier Value Specification

Object identifier values are somewhat different in that they result in a structure being populated in the C or C++ source file.

ASN.1 production: `<name> OBJECT IDENTIFIER ::= <value>`

Generated code: `ASN1OBJID <name> = <value>;`

For example, consider the following declaration:

```
oid OBJECT IDENTIFIER ::= { ccitt b(5) 10 }
```

This would result in the following definition in the C or C++ source file:

```
ASN1OBJID oid = {
   3,
   { 0, 5, 10 }
} ;
```

To populate a variable in a generated structure with this value, the *rtSetOID* utility function can be used (see the section in the run-time API guide for a full description of this function). In addition, the C++ base type for this construct (ASN1TObjId) contains constructors and assignment operators that allow direct assignment of values in this from to the target variable.

The professional version of the ASN1C compiler will generate code for following remaining value definitions.

- SEQUENCE
- SET
- SEQUENCE OF
- SET OF
- CHOICE

NOTE:
SEQUENCE, SET , SEQUENCE Of, SET Of and CHOICE values are available only when the *-tables* option is selected. The values are initialized in a module value initialization function. The format of this function name is as follows:

init_<*ModuleName*>Value (ASN1CTXT* pctxt)

Where <*ModuleName*> would be replaced with the name of the module containing the value specifications.

The only required argument is an initialized context block structure used to hold dynamic memory allocated in the creation of the value structures.

If the value definitions are used in table constraint definitions, then the generated table constraint processing code will handle the initialization of these definitions; otherwise, the initialization function must be called explicitly.

SEQUENCE Value Specification

The mapping of an ASN.1 declaration to a global C or C++ value declaration is as follows:

ASN.1 production: `<name> <SeqType> ::= <value>`

Generated code: `<SeqType> <name>;`

The sequence value will be initialized in the value initialization function.

For example, consider the following declaration:

```
SeqType ::=  SEQUENCE {
             id  INTEGER ,
             name VisibleString
}

value SeqType ::= { id 12, name "abc" }
```

This would result in the following definition in the C or C++ source file:

```
SeqType value;
```

Code generated in value unitization function:

```
value.id = 12;
value.name = "abc";
```

SET Value Specification

SET value code generation is the same as SEQUENCE value code generation.

SEQUENCE OF Value Specification

The mapping of the ASN.1 declaration to a global C or C++ value declaration is as follows:

ASN.1 production: `<name> <SeqOfType> ::= <value>`

Generated code: `<SeqOfType> <name>;`

The sequence of value will be initialized in the value initialization function.

For example, consider the following declaration:

```
SeqOfType ::=  SEQUENCE OF (SIZE(2)) INTEGER

value SeqOfType ::= { 1, 2 }
```

This would result in the following definition in the C or C++ source file:

```
SeqOfType value;
```

Code generated in value initialization function:

```
value.n = 2;
value.element[0] = 1;
value.element[1] = 2;
```

SET OF Value Specification

SET OF value code generation is the same as SEQUENCE OF value code generation.

CHOICE Value Specification

The mapping of ASN.1 declaration to global C or C++ value declaration is as follows:

ASN.1 production: `<name> <ChoiceType> ::= <value>`

Generated code: `<ChoiceType> <name>;`

The choice value will be initialized in the value initialization function.

For example, consider the following declaration:

```
ChoiceType ::=  CHOICE { oid OBJECT IDENTIFIER, id INTEGER }

value ChoiceType ::= id: 1
```

This would result in the following definition in the C or C++ source file:

```
ChoiceType value;
```

Code generated in value initialization function:

```
value.t = T_ChoiceType_id;
value.u.id = 1;
```

CLASS specification

> **NOTE**: Class code generation is available for -tables option only.
>
> This additional code is generated to support the processing required to verify table constraints. This is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

All of the Class code will be generated in a module class header file with the following filename format:

<*ModuleName*>Class.h

In this definition, <*ModuleName*> would be replaced with the name of the ASN.1 module name for this class definition.

C Code generation:

The C struct definition generated to model an ASN.1 class contains member variables for each of the fields within the class.

For each of the following class fields, the corresponding member variable is included in the generated C structure as follows:

For a Value Field:

```
<TypeName> <FieldName>;
```

For TypeField definitions, an encode and decode function pointer and type size field is generated to hold the information of the type for the OpenType. If the *-print* option is selected, a print function pointer is also added.

```
int <FieldName>Size;
int (*encode<FieldName>) (... );
int (*decode<FieldName>) (... );
void (*print<FieldName>) (...);
```

For an Object Field:

```
<ClassName>* <FieldName>;
```

For an ObjectSetField definition, an array of class definitions is generated to hold the list of information objects.

```
<ClassName>* <FieldName>;
```

In each of these definitions:

<FieldName> would be replaced with the name of the field (without the leading '&').
<TypeName> would be replaced with the C type name for the ASN.1 Type.
<ClassName> would be replaced with the C type name of the class for the Information Object.

As an example, consider the following ASN.1 class definition :

```
ATTRIBUTE               ::=     CLASS {
        &Type,
        &id                     OBJECT IDENTIFIER UNIQUE }
WITH SYNTAX {
        WITH SYNTAX &Type ID &id }
```

This would result in the following definition in the C source file:

```
typedef struct ATTRIBUTE {
   int TypeSize;
   int (*encodeType) (ASN1CTXT* , void *, ASN1TagType );
   int (*decodeType) (ASN1CTXT* , void *, ASN1TagType, int );
   ASN1OBJID id;
}
```

C++ Code generation:

The C++ abstract class generated to model an ASN.1 CLASS contains member variables for each of the fields within the class. Derived information object classes are required to populate these variables with the values defined in the ASN.1 information object specification. The C++ class also contains virtual methods representing each of the type fields within the ASN.1 class specification. If the field is not defined to be OPTIONAL in the ASN.1 specification, then it is declared to be abstract in the generated class definition. A class generated for an ASN.1 information object that references this base class is required to implement these abstract virtual methods.

For each of the following CLASS fields, a corresponding member variable is generated in C++ class definition as follows:

For a Value Field definition, the following member variable will be added. Also, an Equals() method will be added if required for table constraint processing.

```
   <TypeName> <FieldName>;

   inline ASN1BOOL idEquals  (<TypeName>* pvalue)
```

For a Type Field definition, a virtual method is added for each encoding rules type to call the generated C encode and decode functions. If -print is specified, a print method is also generated.

```
   virtual int encode<ER><FieldName>
      (ASN1CTXT* pctxt, ASN1TObject& object) { return 0; }

   virtual int decode<ER><FieldName>
      (ASN1CTXT* pctxt, ASN1TObject& object) { return 0; }

   virtual void print<FieldName>
      (ASN1ConstCharPtr name, ASN1TObject& object) {}
```

For an Object Field:

```
   class <ClassName>* <FieldName>;
```

In each of these definitions:

<FieldName> would be replaced with the name of the field (without the leading '&').
<TypeName> would be replaced with the C type name for the ASN.1 Type.
<ClassName> would be replaced with the C type name of the class for the Information Object.
<ER> would be replaced by an encoding rules type (BER, PER, or XER).

As an example, consider the following ASN.1 class definition :

```
ATTRIBUTE ::= CLASS {
        &Type,
        &ParameterType OPTIONAL,
        &id            OBJECT IDENTIFIER UNIQUE }
WITH SYNTAX {
        WITH SYNTAX &Type ID &id }
```

This would result in the following definition in the C++ source file:

```
class EXTERN ATTRIBUTE {
 protected:
   ASN1TObjId id;

   ATTRIBUTE ();

 public:
   virtual int encodeBERType
      (ASN1CTXT* pctxt, ASN1TObject& object) = 0;

   virtual int decodeBERType
      (ASN1CTXT* pctxt, ASN1TObject& object) = 0;

   ASN1BOOL isParameterTypePresent() {
       if(m.ParameterTypePresent) {return TRUE;} else {return FALSE;}
   }
   virtual int encodeBERParameterType
      (ASN1CTXT* pctxt, ASN1TObject& object) { return 0; }

   virtual int decodeBERParameterType
      (ASN1CTXT* pctxt, ASN1TObject& object) { return 0; }

   inline ASN1BOOL idEquals (ASN1TObjId* pvalue)
   {
      return (0 == rtCmpTCOID (&id, pvalue));
   }
} ;
```

This assumes that only BER or DER was specified as the encoding rules type.

First notice that member variables have been generated for the fixed-type fields in the definition. These include the *id* field. Information object classes derived from this definition are expected to populate these fields in their constructors.

Also, virtual methods have been generated for each of the type fields in the class. These include the *Type* fields. The method generated for *Type* is abstract and must be implemented in a derived information object class. The method generated for *ParameterType* field have default implementations that do nothing, is it is a optional field.

Also generated are *Equals* methods for the fixed-type fields. These are used by the generated code to verify that data in a generated structure to be encoded (or data that has just been decoded) matches the table constraint values. This method will be generated only if it is required to check a table constraint.

OPTIONAL keyword:

Fields within a CLASS can be declared to be optional using the OPTIONAL keyword. This indicates that the field is not required in the information object. An additional construct is added to the generated code to indicate whether an optional field is present in the information object or not. This construct is a bit structure placed at the beginning of the generated structure. This structure always has variable name 'm' and contains single-bit elements of the form '<field-name>Present' as follows:

```
struct {
   unsigned <field-name1>Present : 1,
   unsigned <field-name2>Present : 1,
   ...
} m;
```

In this case, the fields included in this construct correspond to only those fields marked as OPTIONAL within the CLASS. If a CLASS contains no optional fields, the entire construct is omitted.

For example, we will change the CLASS in the previous example to make one field optional:

```
ATTRIBUTE               ::=     CLASS {
        &Type OPTIONAL,
        &id                     OBJECT IDENTIFIER UNIQUE }
```

In this case, the following C typedef is generated in C struct or C++ class definition:

```
struct {
    unsigned TypePresent : 1;
 } m;
```

When this structure is populated for encoding, the information object processing code will set *TypePresent* flag accordingly to indicate whether the field is present or not.

In C++ code generation, an additional method is generated for an optional field as follows:

```
ASN1BOOL is<FieldName>Present() {
    if (m.<FieldName>Present) {return TRUE;} else {return FALSE;}
}
```

This function is used to check if the field value is present in an information object definition.

Generation of New ASN.1 Assignments from CLASS Assignments:

During CLASS definition code generation, the following new assignments are created for C or C++ code generation:

1. A new Type Assignment is created for a TypeField's type definition, as follows:

```
_<ClassName>_<FieldName>    ::=    <Type>
```

Here *ClassName* is replaced with name of the Class Assignment and *FieldName* is replaced with name of this field. *Type* is the type definition in CLASS's TypeField.

This type is used as a defined type in the information object definition for an absent value of the TypeField. It is also useful for generation of a value for a related Open Type definition in a table constraint.

2. A new Type Assignment is created for a Value Field or Value Set Field type definition as follows (if the type definition is one of the following: ConstraintedType / ENUMERATED / NamedList BIT STRING / SEQUENCE / SET / CHOICE / SEQUENCE OF / SET OF):

```
_<ClassName>_<FieldName>    ::=    <Type>
```

Here *ClassName* is replaced with the name of the CLASS assignment and *FieldName* is replaced with name of the ValueField or ValueSetField. *Type* is the type definition in the CLASS's ValueField or ValueSetField. This type will appear as a defined type in the CLASS's ValueField or ValueSetField.

This new type assignment is used for compiler internal code generation purpose. It is not required for a user to understand this logic.

3. A new Value Assignment is created for a ValueField's default value definition as follows:

```
_<ClassName>_<FieldName>_default   <Type>  ::=   <Value>
```

Here *ClassName* is replaced with name of the Class Assignment and *FieldName* is replaced with name of this ValueField. *Value* is the default value in Class's ValueField and *Type* is the type in Class's ValueField.

This value is used as a defined value in the information object definition for an absent value of the field. This new value assignment is used for compiler internal code generation purpose. It is not required for user to understand this logic.

ABSTRACT-SYNTAX and TYPE-IDENTIFIER

The ASN.1 ABTRACT-SYNTAX and TYPE-IDENTIFIER classes are useful ASN.1 definitions. These classes are described using the following ASN.1 definitions:

```
TYPE-IDENTIFIER ::= CLASS {
   &id OBJECT IDENTIFIER UNIQUE,
   &Type
}
WITH SYNTAX { &Type IDENTIFIED BY &id }

ABSTRACT-SYNTAX ::= CLASS {
   &id OBJECT IDENTIFIER UNIQUE,
   &Type,
   &property BIT STRING { handles-invalid-encoding(0)} DEFAULT {}
}
WITH SYNTAX {
   &Type IDENTIFIED BY &id [HAS PROPERTY &property]
}
```

The ASN1C compiler generates code for these constructs when they are referenced in the ASN.1 source file that is being compiled. The generated code for these constructs is written to the *Asn1Class.h* and .c/.cpp source files.

Information Object

> NOTE: Information Object code generation is available for -tables option only.
>
> This additional code is generated to support the processing required to verify table constraints. This is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

Information Object code will be generated in a header and source file with a C struct / C++ class to hold the values. The name of the header and source file are of the following format:

<*ModuleName*>Table.h
<*ModuleName*>Table.c/cpp

In this definition, <*ModuleName*> would be replaced with the name of the ASN.1 module in which the information object is defined.

C Code Generation:

For C, a global variable is generated to hold information object definition. This is very similar to the code generated for a value definition.

An example of an information object definition that is derived from the ASN.1 ATTRIBUTE class above is as follows:

```
name ATTRIBUTE  ::=  {
    WITH SYNTAX     VisibleString
    ID              { 0 1 1 } }
```

This results in the generation of the following C constant:

```
ATTRIBUTE name;
```

Code generated in information object initialization function:

```
name.TypeSize = sizeof(_name_Type);
name.encodeType = &asn1E__name_Type;
name.decodeType = &asn1D__name_Type;
name.id.numids = 3;
name.id.subid[0] = 0;
name.id.subid[1] = 1;
name.id.subid[2] = 1;
```

C++ Code Generation:

The C++ classes generated for ASN.1 information objects are derived from the ASN.1 class objects. The constructors in these classes populate the fixed-type field member variables with the values specified in the information object. The classes also implement the virtual methods generated for the information object type fields. All non-optional methods are required to be implemented. The optional methods are only implemented if they are defined in the information object definition.

An example of an information object definition that is derived from the ASN.1 class above is as follows:

```
    name ATTRIBUTE  ::=  {
        WITH SYNTAX     VisibleString
        ID              { 0 1 1 } }
```

This results in the generation of the following C++ class:

```
class EXTERN name : public ATTRIBUTE {
 public:
   name();

   virtual int encodeBERType
      (ASN1CTXT* pctxt, ASN1TObject& object);

   virtual int decodeBERType
      (ASN1CTXT* pctxt, ASN1TObject& object);
} ;
```

The constructor implementation for this class (not shown) sets the fixed type fields (*id*) to the assigned values (*{0 1 1}*). The class also implements the virtual methods for the type field virtual methods defined in the base class. These methods simply call the BER encode or decode method for the assigned type (this example assumes -ber was specified for code generation – other encode rules could have been used as well).

Generated Type Assignments

If the information object contains an embedded type definition, it is extracted from the definition to form a new type to be added to the generated C or C++ code. The format of the new type name is as follows:

```
      _<ObjectName>_<FieldName>
```

where <*ObjectName*> is replaced with the information object name and <*FieldName*> is replaced with the name of the field from within the object.

Information Object Set

NOTE: Information Object Set code generation is available for -tables option only.

This additional code is generated to support the processing required to verify table constraints. This code is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

Table constraint processing code to support Information Object Sets is generated in a header and source file with a C struct / C++ class to hold the values. The name of the header and source file are of the following format:

<*ModuleName*>Table.h
<*ModuleName*>Table.c/cpp

In this definition, <*ModuleName*> would be replaced with the name of the ASN.1 module in which the information object is defined.

C Code Generation:

A C global variable is generated containing an array of values for the ASN.1 CLASS definition. Each structure in the array is the equivalent C structure representing the corresponding ASN.1 information object

As of this writing, a static array is used to hold the objects, but this could be changed to something like a linked list or hash.

An example of an Information Object Set definition that is derived from the ASN.1 ATTRIBUTE class above is as follows:

```
SupportedAttributes  ATTRIBUTE  ::= { name | commonName }
```

This results in the generation of the following C constant:

```
ATTRIBUTE SupportedAttributes[2];
int SupportedAttributes_Size = 2;
```

Code generated in the Information Object Set initialization function:

```
SupportedAttributes[0].TypeSize = sizeof(_name_Type);
SupportedAttributes[0].encodeType = &asn1E__name_Type;
SupportedAttributes[0].decodeType = &asn1D__name_Type;
SupportedAttributes[0].id.numids = 3;
SupportedAttributes[0].id.subid[0] = 0;
SupportedAttributes[0].id.subid[1] = 1;
SupportedAttributes[0].id.subid[2] = 1;

SupportedAttributes[1].TypeSize = sizeof(_commonName_Type);
SupportedAttributes[1].encodeType = &asn1E__commonName_Type;
SupportedAttributes[1].decodeType = &asn1D__commonName_Type;
```

```
SupportedAttributes[1].id.numids = 3;
SupportedAttributes[1].id.subid[0] = 0;
SupportedAttributes[1].id.subid[1] = 1;
SupportedAttributes[1].id.subid[2] = 1;
SupportedAttributes[1].id.subid[3] = 1;
```

C++ Code Generation:

In C++, ASN.1 information object sets are mapped to C++ classes. In this case, a C++ singleton class is generated. This class contains a container to hold an instance of each of the ASN.1 information object C++ objects. As of this writing, a static array is used to hold the objects, but this could be changed to something like a linked list or hash without affecting the public interface to the class. The class also contains an object lookup method for each of the key fields. Key fields are identified in the class as either a) fields that are marked unique, or b) fields that are referenced in table constraints with the '@' notation.

The generated constructor initializes all required values and information objects.

An example of an information object set that uses the information object class defined above is as follows:

```
SupportedAttributes  ATTRIBUTE  ::= { name | commonName }
```

This results in the generation of the following C++ class:

```
class EXTERN SupportedAttributes {
 protected:
   ATTRIBUTE* mObjectSet[2];
   const size_t mNumObjects;
   static SupportedAttributes* mpInstance;
   SupportedAttributes (ASN1CTXT* pctxt);

 public:
   ATTRIBUTE* lookupObject (ASN1TObjId _id);

   static SupportedAttributes* instance(ASN1CTXT* pctxt);
} ;
```

The *mObjectSet* array is the container for the information object classes. These objects are created and this array populated in the class constructor. Note that this is a singleton class (as evidenced by the protected constructor and *instance* methods). Therefore, the object set array is only initialized once the first time the *instance* method is invoked.

The other method of interest is the *lookupObject* method. This was generated for the *id* field because it was identified as a key field. This determination was made because *id* was declared to be UNIQUE in the class definition above. A field can also be determined to be a key field if it is referenced via the '@' notation in a table constraint in a standard type definition. For example, in the following element assignment:

```
argument OPERATION.&Type ({SupportedAttributes}{@opcode})
```

the *opcode* element's *ATTRIBUTE* class field is identified as a key field.

Encode/Decode Function Prototypes

If BER or DER encoding is specified, a BER encode and decode function prototype is generated for each production (DER uses the same form – there are only minor differences between the two types of generated functions). These prototypes are of the following general form:

```
int asn1E_<ProdName> (ASN1CTXT* pctxt,
   <ProdName>* data_p, ASN1TagType tagging);

int asn1D_<ProdName> (ASN1CTXT* pctxt,
   <ProdName>* data_p, ASN1TagType tagging, int length);
```

The prototype with the *asn1E_* prefix is for encoding and the one with *asn1D_* is for decoding. The first parameter is a context variable used for reentrancy. This allows the encoder/decoder to keep track of what it is doing between function invocations.

The second parameter is for passing the actual data variable to be encoded or decoded. This is a pointer to a variable of the generated type.

The third parameter specifies whether implicit or explicit tagging should be used. In practically all cases, users of the generated function should set this parameter to ASN1EXPL (explicit). This tells the encoder to include an explicit tag around the encoded result. The only time this would not be used is when the encoder or decoder is making internal calls to handle implicit tagging of elements.

The final parameter (decode case only) is length. This is ignored when tagging is set to ASN1EXPL (explicit), so users can ignore it for the most part and set it to zero. In the implicit case, this specifies the number of octets to be extracted from the byte stream. This is necessary because implicit indicates no tag/length pair precedes the data; therefore it is up to the user to indicate how many bytes of data are present.

If PER encoding is specified, the format of the generated prototypes is different. The PER prototypes are of the following general form:

```
int asn1PE_<ProdName> (ASN1CTXT* pctxt, <ProdName>[*] value);

int asn1PD_<ProdName> (ASN1CTXT* pctxt, <ProdName>* pvalue);
```

In these prototypes, the prefixes are different (a 'P' character is added to indicate they are PER encoders/decoders), and the tagging argument variables are omitted. In the encode case, the value of the production to be encoded may be passed by value if it is a simple type (for example, BOOLEAN or INTEGER). Structured values will still be passed using a pointer argument.

If XER encoding is specified, a single encode function prototype is generated with the following format:

```
int asn1XE_<ProdName> (ASN1CTXT* pctxt, <ProdName>[*] value,
                       ASN1ConstCharPtr elemName,
                       ASN1ConstCharPtr attrs);
```

This signature includes an element name argument that contains the name of the element to be encoded and attributes argument that can be used to encode an attributes string. There is no C decode function generated for XER and hence no function prototype. Decoding is done using a SAX handler object that is only supported for C++.

Generated C++ Control Class Definition

A control class definition is generated for each defined production in the ASN.1 source file that is determined to be a *Protocol Data Unit* (PDU). By default, any type defined in an ASN.1 source file that is not referenced by any other type is a PDU. This default behavior can be overridden by using a configuration file setting (<*isPDU/*>) or a command-line option (*-pdu)* to explicitly declare that certain types are PDU's.

The generated control class is derived from the *ASN1CType* base class. This class provides a set of common attributes and methods for encoding/decoding ASN.1 messages. It hides most of the complexity of calling the encode/decode functions directly.

BER/DER or PER Class Definition

The general form of the class definition for BER, DER, or PER encoding rules is as follows:

```
class ASN1C_<name> : public ASN1CType {
protected:
   ASN1T_<name>& msgData;
public:
   ASN1C_<name> (ASN1T_<name>& data);
   ASN1C_<name> (
      ASN1MessageBufferIF& msgBuf, ASN1T_<name>& data);

   // standard encode/decode methods (defined in ASN1CType base class):
   // int Encode ();
   // int Decode ();

   // stream encode/decode methods:
   int EncodeTo (ASN1MessageBufferIF& msgBuf);
   int DecodeFrom (ASN1MessageBufferIF& msgBuf);
 } ;
```

The name of the generated class is *ASN1C_*<*name*> where <*name*> is the name of the production. The only defined attribute is a protected variable reference named *msgData* of the generated type.

Two constructors are generated. The first is for stream operations and allows the control class to be created using only a reference a variable of the generated type.

The *EncodeTo* and *DecodeFrom* methods can then be used to encode or decode directly to and from a stream. The << and >> stream operators can be used as well.

The second constructor is the legacy form that allows a message buffer to be associated with a data variable at the time of creation. The *Encode* and *Decode* methods defined in the *ASN1CType* base class can be used with this construction form to encode and decode to the associated buffer.

The constructor arguments are a reference to an *ASN1MessageBufferIF* (message buffer interface) type and a reference to an *ASN1T_*<*name*> type. The message buffer interface argument is a reference to an abstract message buffer or stream class. Implementations of the interface class are available for BER/DER, PER, or XER encode or decode message buffers or for a BER or XER encode or decode stream.

The *ASN1T_<name>* argument is used to specify the data variable containing data to be encoded or to receive data on a decode call. The procedure for encoding is to declare a variable of this type, populate it with data, and then instantiate the *ASN1C_<name>* object to associate a message buffer object with the data to be encoded. The *Encode* or *Encode To* method can then be called to encode the data. On the decode side, a variable must be declared and passed to the constructor to receive the decoded data.

Note that the *ASN1C_* class declarations are only required in the application code as an entry point for encoding or decoding a top-level message (or Protocol Data Unit – PDU). As of ASN1C version 5.6, control classes are only generated for ASN.1 types that are determined to be PDU's. A type is determined to be a PDU if it is referenced by no other types. This differs from previous version of ASN1C where control classes were generated for all types. This default behavior can be overridden by using a configuration file entry or the -pdu command-line switch to explicitly declare the PDU types. The *<isPDU/>* flag is used to declare a type to be a PDU in a configuration file. An example of this is as follows:

```
<asn1config>
  <module>
    <name>H323-MESSAGES</name>
    <production>
      <name>H323-UserInformation</name>
      <isPDU/>
    </production>
  </module>
</asn1config>
```

This will cause only a single *ASN1C_* control class definition to be added to the generated code for the *H323-UserInformation* production.

If the module contains no PDUs (i.e,. contains support types only), the <noPDU/> empty element can be specified at the module level to indicate that no control classes should be generated for the module.

XER Class Definition

For the XML encoding rules (XER), the generated class definition is as follows:

```
class ASN1C_<name> :
   public ASN1CType, ASN1XERSAXHandler
{
protected:
   ASN1T_<name>& msgData;
   ...  additional control variables
public:
   ASN1C_<name> (ASN1T_<name>& data);
   ASN1C_<name> (
      ASN1MessageBufferIF& msgBuf, ASN1T_<name>& data);

   // standard encode/decode methods (defined in ASN1CType base class):
   // int Encode ();
   // int Decode ();

   // stream encode/decode methods:
```

```
   int EncodeTo (ASN1MessageBufferIF& msgBuf);
   int DecodeFrom (ASN1MessageBufferIF& msgBuf);

   // SAX Content Handler Interface

   virtual void startElement
      (const XMLCh* const uri,
       const XMLCh* const localname,
       const XMLCh* const qname,
       const Attributes& attrs);

   virtual void characters
      (const XMLCh* const chars, const unsigned int length);

   virtual void endElement
      (const XMLCh* const uri,
       const XMLCh* const localname,
       const XMLCh* const qname);

} ;
```

The main differences between the BER/DER/PER control class definition and this are:

1. The class generated for XER inherits from the *ASN1XERSAXHandler* base class, and
2. The class implements the standard SAX content handler methods.

This allows an object of this class to be registered as a SAX content handler with any SAX-compliant XML parser. The parser would be used to read and parse XML documents. The methods generated by ASN1C would then receive the parsed data via the SAX interface and use the results to populate the data variables with the decoded data.

Generated Methods

For each production, an *EncodeFrom* and *DecodeTo* method is generated within the generated class structure. These are standard methods that initialize context information and then call the generated C-like encode or decode function. If the generation of print functions was specified (by including –print on the compiler command line), a *Print* method is also generated that calls the C print function.

For XER, additional methods are generated to implement a SAX content handler interface to an XML parser. This includes a *startElement*, *characters*, and *endElement* method. An *init* and *finalize* method may also be generated to initialize a variable prior to parsing and to complete population of a variable with decoded data.

Generated Information Object Table Structures

Information Objects and Classes are used to define multi-layer protocols in which “holes” are defined within ASN.1 types for passing message components to different layers for processing. These items are also used to define the contents of various messages that are allowed in a particular exchange of messages. The ASN1C compiler extracts the types involved in these message exchanges and generates encoders/decoders for them. The “holes” in the types are accounted for by adding open type holders to the generated structures. These open type holders consist of a byte count and pointer for storing information on an encoded message fragment for processing at the next level.

The ASN1C compiler is capable of generating code in one of two forms for information in an object specification:

1. Simple form: in this form, references to variable type fields within standard types are simply treated as open types and an open type placeholder is inserted.

2. Table form: in this form, all of the classes, objects, and object sets within a specification result in the generation of code for parsing and formatting the information field references within standard type structures.

The second form is selected by specifying the –tables command line option.

To better understand the support in this area, the individual components of Information Object specifications are examined. We begin with the "CLASS" specification that provides a schema for Information Object definitions. A sample class specification is as follows:

```
OPERATION ::= CLASS {
   &operationCode          CHOICE { local  INTEGER,
                                    global OBJECT IDENTIFIER }
   &ArgumentType,
   &ResultType,
   &Errors                 ERROR       OPTIONAL
}
```

Users familiar with ASN.1 will recognize this as a simplified definition of the ROSE OPERATION MACRO using the Information Object format. When a class specification such as this is parsed, information on its fields is maintained in memory for later reference. In the simple form of code generation, the class definition itself does not result in the generation of any corresponding C or C++ code. It is only an abstract template that will be used to define new items later on in the specification. In the table form, if C++ is specified, an abstract base class is generated off of which other classes are derived for information object specifications.

Fields from within the class can be referenced in standard ASN.1 types. It is these types of references that the compiler is mainly concerned with. These are typically "header" types that are used to add a common header to a variety of other message body types. An example would be the following ASN.1 type definition for a ROSE invoke message header:

```
Invoke ::= SEQUENCE {
   invokeID     INTEGER,
   opcode       OPERATION.&operationCode,
   argument     OPERATION.&ArgumentType
}
```

This is a very simple case that purposely omits a lot of additional information such as Information Object Set constraints that are typically a part of definitions such as this. The reason this information is not present is because we are just interested in showing the items that the compiler is concerned with. We will use this type to demonstrate the simple form of code generation. We will then add table constraints and discuss what changes when the –tables command line options is used.

The opcode field within this definition is an example of a **fixed type** field reference. It is known as this because if you go back to the original class specification, you will see that operationCode is defined to be of a specific type (namely a choice between a local and global value). The generated typedef for this field will contain a reference to the type from the class definition.

The argument field is an example of a **variable type** field.. In this case, if you refer back to the class definition, you will see that no type is provided. This means that this field can contain an instance of any encoded type (note: in practice, table constraints can be used with Information Object Sets to limit the message types that can be placed in this field). The generated typedef for this field contains an "open type" (ASN1OpenType) reference to hold a previously encoded component to be specified in the final message.

Simple Form Code Generation

In the simple form of information object code generation, the *Invoke* type above would result in the following C or C++ typedefs being generated:

```
typedef struct Invoke ::= SEQUENCE {
   ASN1INT        invokeID;
   OPERATION_operationCode opcode;
   ASN1OpenType   argument;
}
```

The following would be the procedure to add the Invoke header type to an ASN.1 message body:

1. Encode the body type
2. Get the message pointer and length of the encoded body
3. Plug the pointer and length into the "numocts" and "data" items of the argument open type field in the Invoke type variable.
4. Populate the remaining Invoke type fields.
5. Encode the Invoke type to produce the final message.

In this case, the amount of code generated to support the information object references is minimal. The amount of coding required by a user to encode or decode the variable type field elements, however, can be rather large. This is a tradeoff that exists between using the compiler generated table constraints solution (as we will see below) and using the simple form.

Table Form Code Generation

If we now add table constraints to our original type definition, it might look as follows:

```
Invoke ::= SEQUENCE {
  invoke     IDINTEGER,
  opcode     OPERATION.&operationCode ({My-ops}),
  argument   OPERATION.&ArgumentType ({My-ops}{@opcode})
}
```

The "{My-ops}" constraint on the opcode element specifies an information object set (not shown) that constrains the element value to one of the values in the object set. The {My-ops}{@opcode} constraint on the argument element goes a step further – it ties the type of the field to the type specified in the row that matches the given opcode value.

ASN1C generates an in-memory table (either an array or a list of structures) for each of the items in the information object sets defined in a specification. In the example above, a table would be generated for the *My-ops* information object set. The code generated for the type would then use this table to verify that the given items in a structure that reference this table match the constraints.

The C or C++ type generated for the SEQUENCE above when –tables is specified would be as follows:

```
typedef struct Invoke {
   ASN1INT        invokeID;
   OPERATION_operationCode opcode;
   ASN1Object     argument;
} Invoke;
```

This is almost identical to the type generated in the simple case. The difference is the *ASN1Object* type (or *ASN1TObject* for C++) that is used instead of *ASN1OpenType*. This type is defined in the *asn1type.h* run-time header file as follows:

```
typedef struct ASN1Object {
   ASN1OpenType  encoded;
   void*         decoded;
   ASN1INT       index;
}
```

This holds the value to be encoded or decoded in both encoded or decoded form. The way a user uses this to encode a value of this type is as follows:

1. Populate a variable of the type to be used as the argument to the invoke type.
2. Plug the address of this variable into the *decoded* void pointer in the structure above.
3. Populate the remaining Invoke type fields.
4. Encode the Invoke type to produce the final message.

Note that in this case, the intermediate type does not need to be manually encoded by the user. The generated encoder has logic built-in to encode the complete message using the information in the generated tables.

Additional Code Generated with -tables option

When the –tables command line option is used, additional code is generated to support the additional processing required to verify table constraints. This code varies depending on whether C or C++ code generation is selected. The C++ code is designed to take advantage of the object-oriented capabilities of C++. These capabilities are well suited for modeling the behavior of information objects in practice. The following subsections describe the code generated for each of these languages.

The code generated to support these constraints is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

C Code Generation

For C, code is generated for the Information Object Sets defined within a specification in the form of a global array of structures. Each structure in the array is an equivalent C structure representing the corresponding ASN.1 information object.

Additional encode and decode functions are also generated for each type that contains table constraints. These functions have the following prototypes:

BER/DER:

```
int asn1ETC_<ProdName> (ASN1CTXT* pctxt, <ProdName>* pvalue);

int asn1DTC_<ProdName> (ASN1CTXT* pctxt, <ProdName>* pvalue);
```

PER:

```
int asn1PETC_<ProdName> (ASN1CTXT* pctxt, <ProdName>* pvalue);

int asn1PDTC_<ProdName> (ASN1CTXT* pctxt, <ProdName>* pvalue);
```

The purpose of these functions is to verify the fixed values within the table constraints are what they should be and to encode or decode the open type fields using the encoder or decoder assigned to the given table row. Calls to these functions are automatically built into the standard encode or decode functions for the given type. They should be considered hidden functions not for use within an application that uses the API.

C++ Code Generation

For C++, code is generated for ASN.1 classes, information objects, and information object sets. This code is then referenced when table constraint processing must be performed.

Each of the generated C++ classes builds on each other. First, the classes generated that correspond to ASN.1 CLASS definitions form the base class foundation. Then C++ classes derived from these base classes corresponding to the information objects are generated. Finally, C++ singleton classes corresponding to the information object sets are generated. Each of these classes provides a container for a collection of C++ objects that make up the object set.

Additional encode and decode functions are also generated as they were in the C code generation case for interfacing with the object definitions above. These functions have the following prototypes:

BER/DER:

```
int asn1ETC_<ProdName> (ASN1CTXT* pctxt,
                        <ProdName>* pvalue,
                        <ClassName>* pobject);

int asn1DTC_<ProdName> (ASN1CTXT* pctxt,
                        <ProdName>* pvalue,
                        <ClassName>* pobject);
```

PER:

```
int asn1PETC_<ProdName> (ASN1CTXT* pctxt,
                         <ProdName>* pvalue,
                         <ClassName>* pobject);

int asn1PDTC_<ProdName> (ASN1CTXT* pctxt,
                         <ProdName>* pvalue,
                         <ClassName>* pobject);
```

These prototypes are identical to the prototypes generated in C code generation case except for the addition of the *pobject* argument. This argument is for a pointer to the information object that matches the key field value for a given encoding. These functions have different logic for processing Relative and Simple table constraints. The logic associated with each case is as follows:

On the encode side:
Relative Table Constraint:

1. The *lookupObject* method is invoked on the object set instance to find the class object for the data in the populated type variable to be encoded.

2. If a match is found, the table constraint encode function as defined above is invoked. This function will verify all fixed type values match what is defined in the information object definition and will encode all type fields and store the resulting encoded data in the *ASN1TObject.encoded* fields.

3. If a match is not found and the information object set is not extensible, then a table constraint error status will be returned. If the information object set is extensible, a normal status is returned.

Simple Table Constraint:

1. This function will verify all the fixed type values match what is defined in the table constraint information object set. If an element value does not exist in the table (i.e. the information object set) and the object set is not extensible, then a table constraint violation exception will be thrown.

The normal encode logic is then performed to encode all of the standard and open type fields in the message.

On the decode side, the logic is reversed:

The normal decode logic is performed to populate the standard and open type fields in the generated structure.
Relative Table Constraint:

1. The *lookupObject* method is invoked on the decoded key field value to find an object match.

2. If a match is found, the table constraint decode function as defined above is invoked. This function will verify all fixed type values match what is defined in the information object definition and will fully decode all type fields and store pointers to the decoded type variables in the *ASN1TObject.decoded* fields.

3. If a match is not found and the information object set is not extensible, then a table constraint error status will be returned. If the information object set is extensible, a normal status is returned.

Simple Table Constraint:

1. This function will verify all the fixed type values match what is defined in the table constraint object set. If an element value does not exist in the table (i.e. the information object set) and the object set is not extensible, then a table constraint violation exception will be thrown.

General Procedure for Table Constraint Encoding

The general procedure to encode an ASN.1 message with table constraints is the same as without table constraints. The only difference is in the open type data population procedure. The -tables option will cause *ASN1TObject* fields to be inserted in the generated code instead of *Asn1OpenType* declarations.

Refer to the BER/DER/PER encoding procedure for further information.

The procedure to populate the value for an *ASN1TObject* item is as follows:

1. Check the ASN.1 specification or generated C code for the type of the type field value in the information object set that corresponds to the selected key field value.

2. Create a variable of that type and assign a pointer to it to the *Asn1Object.decoded* member variable as void*.

3. Follow the common BER/PER/DER encode procedure.

A complete example showing how to assign an open type value is as follows:

```
Test DEFINITIONS ::= BEGIN

   ATTRIBUTE               ::=     CLASS {
        &Type,
        &id                        OBJECT IDENTIFIER UNIQUE }
   WITH SYNTAX {
        WITH SYNTAX &Type ID &id }

   name ATTRIBUTE  ::=  {
        WITH SYNTAX     VisibleString
        ID              { 0 1 1 } }

   name ATTRIBUTE  ::=  {
        WITH SYNTAX     INTEGER
        ID              { 0 1 2 } }

   SupportedAttributes  ATTRIBUTE  ::= { name | commonName }

   Invoke ::= SEQUENCE {
      opcode   ATTRIBUTE.&id   ({SupportedAttributes}),
      argument ATTRIBUTE.&Type ({SupportedAttributes}{@opcode})
   }

END
```

In the above example, the *Invoke* type contains a table constraint. Its element *opcode* refers to the ATTRIBUTE *id* field and *argument* element refers to the ATTRIBUTE *Type* field. The *opcode* element is an index element for the *Invoke* type's table constraint. The *argument* element is an open type whose type is determined by the *opcode* value. In this example, *opcode* is the key field.

The *opcode* element can have only two possible values: { 0 1 1 } or { 0 1 2 }. If the *opcode* value is { 0 1 1} then *argument* will have a *VisibleString* value and if the *opcode* value is { 0 1 2 } then *argument* will have an INTEGER value. Any other value of the *opcode* element will be violation of the Table Constraint.

If the *SupportedAttributes* information object set was extensible (indicated by a ",..." at the end of the definition), then *the argument* element may have a value of a type that is not in the defined set. In this case, if the index element value is outside the information object set, then the *argument* element will be assumed to be an *Asn1OpenType*. The *Invoke* type encode function call will use the value from *argument.encoded.data* field (i.e. it will have to be pre-encoded because the encode function will not be able to determine from the table constraint how to encode it).

A **C++ program** fragment that could be used to encode an instance of the *Invoke* type is as follows:

```
#include TestTable.h         // include file generated by ASN1C

main ()
{
    const ASN1OCTET* msgptr;
    ASN1OCTET msgbuf[1024];
    int       msglen;

    // step 1: construct ASN1C C++ generated class.
    // this specifies a static encode message buffer

    ASN1BEREncodeBuffer encodeBuffer (msgbuf, sizeof(msgbuf));

    // step 2: populate msgData structure with data to be encoded

    ASN1T_Invoke msgData;
    ASN1C_Invoke invoke (encodeBuffer, msgData);

    msgData.opcode.numids = 3;
    msgData.opcode.subid[0] = 0;
    msgData.opcode.subid[1] = 1;
    msgData.opcode.subid[2] = 1;
    ASN1VisibleString argument = "objsys";
    msgData.argument.decoded = (void*) &argument;
    // note: opcode value is  {0 1 1 }, so argument must be
    // ASN1VisibleString type

    // step 3: invoke Encode method

    if ((msglen = invoke.Encode ()) > 0) {
       // encoding successful, get pointer to start of message
       msgptr = encodeBuffer.getMsgPtr();
    }
    else
      error processing...
}
```

The encoding procedure for C requires one extra step. This is a call to the module initialization functions after context initialization is complete. All module initialization functions for all modules in the project must be invoked. The module initialization function definitions can be found in the <*ModuleName*>*Table.h* file.

The format of each module initialization function name is as follows:

```
void <ModuleName>_init (ASN1CTXT* pctxt)
```

Here *ModuleName* would be replaced with name of the module.

A **C program** fragment that could be used to encode the *Invoke* record defined above is as follows:

```
#include TestTable.h         /* include file generated by ASN1C */
```

```
int main ()
{
    ASN1OCTET msgbuf[1024], *msgptr;
    int       msglen;
    ASN1CTXT  ctxt;
    Invoke    invoke;  /* typedef generated by ASN1C */

    /* Step 1: Initialize the context and set the buffer pointer */

    if (rtInitContext (&ctxt) != ASN_OK) {
       /* initialization failed, could be a license problem */
       printf ("context initialization failed (check license)\n");
       return -1;
    }

    xe_setp (&ctxt, msgbuf, sizeof(msgbuf));

    /* step 2: call module initialization functions */

    Test_init (&ctxt);

    /* Step 3: Populate the structure to be encoded */

    msgData.opcode.numids = 3;
    msgData.opcode.subid[0] = 0;
    msgData.opcode.subid[1] = 1;
    msgData.opcode.subid[2] = 1;
    //note: opcode value is  {0 1 1 }, so argument must be
    //ASN1VisibleString type
    ASN1VisibleString argument = "objsys";
    msgData.argument.decoded = (void*) &argument;
    ...

    /* Step 4: Call the generated encode function */

    msglen = asn1E_Invoke (&ctxt, &invoke, ASN1EXPL);

    /* Step 5: Check the return status (note: the test is */
    /* > 0 because the returned value is the length of the */
    /* encoded message component)..*/

    if (msglen > 0) {

      /* Step 6: If encoding is successful, call xe_getp to */
      /* fetch a pointer to the start of the encoded message.*/

      msgptr = xe_getp (&ctxt);
      ...
    }
    else
      error processing...
```

```
}
```

General Procedure for Table Constraint decoding

The general procedure to decode an ASN.1 message with table constraints is the same as without table constraints. The only difference will exist in the decoded data for open type fields within the message. In this case, the *Asn1Object* / *Asn1TObject*'s *decoded* member variable will contain the original decoded type and *the encoded* member variable will contain the original data in encoded form.

Refer to the BER/DER/PER decoding procedure for further information.

The procedure to retrieve the value for open type fields is as follow:

1. Check the possible Type in the Information Object Set from index element value.
2. Assign or cast the *Asn1Object.decoded* member variable (void*) to the result type.
3. The *Asn1Object.encoded* field will hold the data in encoded form.

For the above complete example, the *Invoke* type's *argument* element will be decoded as one of the types in the *SupportedAttributes* information object set (i.e. either as a *VisibleString* or *INTEGER* type). If the *SupportedAttributes* information object set is extensible, then the *argument* element may be of a type not defined in the set. In this case, the decoder will set the *Asn1Object.encoded* field as before but the *Asn1Object.decoded* field will be NULL indicating the value is of an unknown type.

A **C++ program** fragment that could be used to decode the *Invoke* example is as follows:

```
#include Test.h         // include file generated by ASN1C

main ()
{
    ASN1OCTET msgbuf[1024];
    ASN1TAG   msgtag;
    int       msglen, status;

    /* step 1: logic to read message into msgbuf */
    ...

    /* step 2: create decode buffer and msg data type */

    ASN1BERDecodeBuffer decodeBuffer (msgbuf, len);

    ASN1T_Invoke msgData;
    ASN1C_Invoke invoke (decodeBuffer, msgData);

    /* step 3: call decode function */

    if ((status = invoke.Decode ()) == ASN_OK)
    {
         // decoding successful, data in msgData
         // use key field value to set type of message data
         ASN1OBJID oid1[] = { 3, { 0, 1, 1 }};
```

```
            ASN1OBJID oid2[] = { 3, { 0, 1, 2 }};
            if (msgData.opcode == oid1) {
               // argument is a VisibleString
               ASN1VisibleString* pArg =
                  (ASN1VisibleString*) msgData.argument.decoded;
               ...
            }
            else if (msgData.opcode == oid2) {
               // argument is an INTEGER
               ASN1INT arg = (ASN1INT) *msgData.argument.decoded;
               ...
            }
         }
         else {
            // error processing
         }
```

In this case, the type of the decoded argument can be determined by testing the key field value. In the example as shown, the *SupportedAttributes* information object set is not extensible, therefore, the type of the argument must be one of the two shown. If the set were extensible (indicated by a ",..." in the definition), then it is possible that an unknown *opcode* could be received which would mean the type can not be determined. In this case, the original encoded message data would be present in *msgData.argument.encoded* field and it would be up to the user to determine how to process it.

The decoding procedure for C requires one additional step. This is a call to the module initialization functions after context initialization is complete. All module initialization functions for all modules in the project must be invoked. The module initialization function definitions can be found in the *<ModuleName>Table.h* file.

A **C program** fragment that could be used to decode the *Invoke* example is as follows:

```
#include TestTable.h         // include file generated by ASN1C

main ()
{
    ASN1OCTET msgbuf[1024];
    ASN1TAG   msgtag;
    int       msglen;
    ASN1CTXT  ctxt;
    Invoke    invoke;
    ASN1OBJID oid1[] = { 3, { 0, 1, 1 }};
    ASN1OBJID oid2[] = { 3, { 0, 1, 2 }};

    .. logic to read message into msgbuf ..

    /* Step 1: Initialize a context variable for decoding */

    if (rtInitContext (&ctxt) != ASN_OK) {
       /* initialization failed, could be a license problem */
       printf ("context initialization failed (check license)\n");
       return -1;
    }
```

```
    xd_setp (&ctxt, msgbuf, 0, &msgtag, &msglen);

    /* step 2: call module initialization functions */

    Test_init (&ctxt);

    /* Step 3: Call decode function */

    status = asn1D_Invoke (&ctxt, &invoke, ASN1EXPL, 0);

    /* Step 4: Check return status */

    if (status == ASN_OK)
    {
         /* process received data in 'invoke' variable */
         if (rtCmpTCOID (&invoke.opcode, &oid1) == 0) {
            /* argument is a VisibleString */
            ASN1VisibleString* pArg =
               (ASN1VisibleString*) msgData.argument.decoded;
            ...
         }
         else if (rtCmpTCOID (&invoke.opcode, &oid2) == 0) {
            /* argument is an INTEGER */
            ASN1INT arg = (ASN1INT) *msgData.argument.decoded;
            ...
         }

         /* Remember to release dynamic memory when done! */
         ASN1MEMFREE (&ctxt);
    }
    else
        error processing...
    }
}
```

Generated C Source Files

By default, the ASN1C compiler generates the following set of .c source files for a given ASN.1 module (note: the name of the module would be substituted for <moduleName>):

`<moduleName>.c`	common definitions and functions (for example, asn1Free_<type>) and/or global value constant definitions.
`<moduleName>Enc.c`	encode functions (asn1E_<type>)
`<moduleName>Dec.c`	decode functions (asn1D_<type>)

If additional options are used (such as –genPrint, -genCopy, etc), additional files will be generated:

`<moduleName>Copy.c`	copy functions, generated if –genCopy is specified
`<moduleName>Print.c`	print functions, generated if –genPrint is specified
`<moduleName>Compare.c`	comparison functions, generated if –genCompare is specified
`<moduleName>PrtToStr.c`	print-to-string functions, generated if –genPrtToStr is specified
`<moduleName>PrtToStrm.c`	print-to-stream functions, generated if –genPrtToStrm is specified
`<moduleName>Table.c`	table constraint functions, generated if –genTable option is specified

If *–genCopy, -genPrint*, etc have a filename parameter then the code will be written to the given file instead of the default one. If the *–cfile <filename>* option is used and *–genCopy, -genPrint*, etc options do not have parameters then all code will be placed in one source file with name <filename>.

Maximum Lines per File

In each of the cases above, it is possible to specify an approximate maximum number of lines that each of the generated .c files may contain. This is done using the *-maxlines* option. If *-maxlines* is specified with no parameter, a default maximum number of lines (50,000) will be set; otherwise, the given value will be used.

If the given maximum lines limit is surpassed in a file, a new file will be started with an "_1" appended, for example *<moduleName>Enc_1.c*. Additional files will be numbered sequentially if necessary (_2, _3, etc.). Note that this limit is a lower threshold and not exact. A complete compilation unit (for example, a function) will not be split because of this threshold. The way it works is the threshold is checked before the output of a compilation unit. If it is found to be exceeded, a new file is started at that time. Therefore, a user should plan for a reserve to be in place above the limit to compensate for this overflow.

The reason for having this limit is because some C/C++ compilers have problems with very large .c files. For example, one product will not allow the debugger to work on lines in a file over the 64k threshold.

Use of the -maxcfiles Option

The *–maxcfiles* option allows generation of more compact code by putting each encode, decode, copy, compare, etc function into a separate file. This allows the linker to link in only the required functions as opposed to all functions in a compiled object module. This option might be useful for applications that have minimal space requirements (for example, embedded systems).

> **NOTE**: Some sophisticated linkers have the capability to pull individual functions out of an object module directly for final inclusion in the target executable or shared object file. In this case, the -maxcfiles option does not provide any advantage in reducing the size of the application program.

To achieve the best results it is necessary to put all compiled object files into an object library (.a or .lib file) and include this library in the link command. The *–genMake* option when used in conjunction with *–maxcfiles* will generate a make-file that will compile each of the generated files and add them to a library with a name based on the name of the ASN.1 module being compiled (<*moduleName*>*.lib* for Windows or *lib*<*moduleName*>*.a* for *NIX).

The format of each generated .c file name is as follows:

```
asn1<suffix>_<prodname>.c
```

where <suffix> depends on encoding rules and function type (encode, decode, free, etc.)
<prodname> is the ASN.1 production name.

For example, consider one type definition within the *Employee.asn* ASN.1 specification:

```
Employee DEFINITIONS ::= BEGIN
...
Name ::= [APPLICATION 1] IMPLICIT SEQUENCE {
   givenName            IA5String,
   initial              IA5String,
   familyName           IA5String
}
...
END
```

By default, the following .c files would be generated (note: this assumes no additional code generation options were selected):

```
Employee.c
EmployeeEnc.c
EmployeeDec.c
```

If *-maxcfiles* was selected as in the following command line:

The ASN1C with –maxcfiles option will generate the following files:

```
asn1c employee.asn -c -ber -trace –maxcfiles
```

the following .c files for this type would be generated for the *Name* type:

```
asn1D_Name.c
asn1E_Name.c
```

These contain the functions to decode *Name* and encode *Name* respectively. Similar files would be generated for the other productions in the module as well.

Generated C++ files

In general, the generation logic for C++ is similar to the logic for C. Instead of the .c file extension, .cpp is used:

<moduleName>.cpp Common definitions and functions (for example, asn1Free_<type>) and/or global

	value constant definitions. This file also contains constructors, destructors and all methods for ASN1C_<Type> and ASN1T_<Type> control classes.
`<moduleName>Enc.cpp`	C encode functions and C++ encode methods.
`<moduleName>Dec.cpp`	C decode functions and C++ decode methods.

If additional options are used (such as *–genPrint*, *-genCopy*, etc), additional files will be generated:

`<moduleName>Copy.cpp`	copy functions, generated if –genCopy is specified
`<moduleName>Print.cpp`	print functions, generated if –genPrint is specified
`<moduleName>Compare.cpp`	comparison functions, generated if –genCompare is specified
`<moduleName>PrtToStr.cpp`	print-to-string functions, generated if –genPrtToStr is specified
`<moduleName>PrtToStrm.cpp`	print-to-stream functions, generated if –genPrtToStrm is specified
`<moduleName>Table.cpp`	table constraint functions, generated if –genTable option is specified

The *–maxcfiles* option for C++ works very similar to how it works for C. The only differences are a few additional files are generated and the .cpp extension is used instead of .c. Additional files are generated to hold *ASN1C_<Type>* and *ASN1T_<Type>* control classes. The format of the filenames of these files is as follows:

```
asn1<suffix>_<prodname>.cpp
ASN1C_<prodname>.cpp
ASN1T_<prodname>.cpp
```

where	<suffix>	depends on encoding rules and function type (encode, decode, free, etc.)
	<prodname>	is the ASN.1 production name.

For the example presented previously in the C Files section, the following files would be generated for the *Name* production in the *Employee.asn* file:

```
asn1D_Name.cpp
asn1E_Name.cpp
ASN1T_Name.cpp
ASN1C_Name.cpp
```

These contain the functions to decode *Name* and encode *Name* respectively. The *ASN1T_Name.cpp* file contains the type class methods, and the *ASN1C_Name.cpp* files contains the control class methods. Note that not all productions have a control class (only PDU types do for BER or PER) therefore the *ASN1C_<type>.cpp* file may not be generated.

Similar files would be generated for the other productions in the module as well.

Note that for C++, the code reduction effect is less than that for pure C. This is because most of the linkers cannot omit virtual methods even if they are not being used by the application. These virtual methods refer to separate C functions and these functions are being linked into the application even if they are not actually used. But, still, the size of the final application created with *–maxcfiles* option should be less than the size of the application created without this option.

Generated C/C++ files and the –compat Option

ASN1C 5.6 and below did not generate separate files for common definitions, encode and decode functions (*<moduleName>.c/.cpp, <moduleName>Enc.c/.cpp, <moduleName>Dec.c/.cpp*). All code was generated in a single file with the name *<moduleName>.c/.cpp*. If it is necessary to maintain this behavior then use the *–compat 5.6* option.

Also, the *–cfile* option's behavior is slightly changed in ASN1C 5.7. In 5.6 and below, the *–cfile* option did not have any effect for files containing copy, print, compare, etc functions. For ASN1C 5.7, *–cfile* causes everything to be output to one file (the exception is if specific filename parameters are specified with *–genPrint*, *-genCopy*, etc). Once again, to maintain the previous behavior the *–compat 5.6* option can be used.

General Procedures for Encoding and Decoding

Encoding functions and methods generated by the ASN1C compiler are designed to be similar in use across the different encoding rule types. In other words, if you have written an application to use the Basic Encoding Rules (BER) and then later decide to use the Packed Encoding Rules (PER), it should only be a simple matter of changing a few function calls to accomplish the change. Procedures for such things as populating data for encoding, accessing decoded data, and dynamic memory management are the same for all of the different encoding rules.

This section describes common procedures for encoding or decoding data that are applicable to any of the different encoding rules. Subsequent sections will then describe what will change for the different rules.

Dynamic Memory Management

The ASN1C run-time uses specialized dynamic memory functions to improve the performance of the encoder/decoder. It is imperative that the user understands how this methodology works in order to avoid memory problems in their applications. ASN1C also provides the capability to plug-in a different memory management scheme at two different levels - the high level API called by the generated code and the low level API that provides the core memory managment functionality.

The ASN1C Default Memory Manager

The default ASN1C run-time memory manager uses an algorithm called the nibble-allocation algorithm. What this means is that large blocks of memory are allocated up front and then split up to provide memory for smaller allocation requests. This reduces the number of calls required to the C *malloc* and *free* functions. These functions are very expensive in terms of performance.

The large blocks of memory are tracked through the ASN.1 context block (ASN1CTXT) structure. For C, this means that an initialized context block is required for all memory allocations and deallocations. All allocations are done using this block as an argument to routines such as *rtMemAlloc*. All memory can be released at once when a user is done with a structure containing dynamic memory items by calling *rtMemFree*. Other functions are available for doing other dynamic memory operations as well. See the *C/C++ Run-time Reference Manual* for details on these.

High Level Memory Management API

The high-level memory management API consists of C macros and functions called in gemerated code and/or in application programs to allocate and free memory within the ASN1C run-time.

At the top level are a set of macro definitions that begin with the prefix *rtMem*. These are mapped to a set of similar functions that begin with the prefix *rtMemHeap*. A table showing this basic mapping is as follows:

Macro	Function	Description
rtMemAlloc	*rtMemHeapAlloc*	Allocate memory
rtMemAllocZ	*rtMemHeapAllocZ*	Allocate and zero memory
rtMemRealloc	*rtMemHeapRealloc*	Reallocate memory
rtMemFree	*rtMemHeapFreeAll*	Free all memory in context

Macro	Function	Description
rtMemFreePtr	*rtMemHeapFreePtr*	Free a specific memory block

See the *ASN1C C/C++ Common Runtime Reference Manual* for further details on these functions and macros.

It is possible to replace the high-level memory allocation functions with functions that implement a custom memory management scheme. This is done by implementing some (or all) of the C *rtMemHeap* functions defined in the following interface (note: a default implementation is shown that replaces the ASN1C memory manager with direct calls to the standard C run-time memory management functions):

```
#include <stdlib.h>
#include "rtMemory.h"

/* Create a memory heap */
int rtMemHeapCreate (void** ppvMemHeap) {
   return ASN_OK;
}

/* Allocate memory */
void* rtMemHeapAlloc (void** ppvMemHeap, int nbytes) {
   return malloc (nbytes);
}

/* Allocate and zero memory */
void* rtMemHeapAllocZ (void** ppvMemHeap, int nbytes) {
   void* ptr = malloc (nbytes);
   if (0 != ptr) memset (ptr, 0, nbytes);
   return ptr;
}

/* Free memory pointer */
void rtMemHeapFreePtr (void** ppvMemHeap, void* mem_p) {
   free (mem_p);
}

/* Reallocate memory */
void* rtMemHeapRealloc (void** ppvMemHeap, void* mem_p, int nbytes_) {
   return realloc (mem_p, nbytes_);
}

/* Clears heap memory (frees all memory, reset all heap's variables) */
void rtMemHeapFreeAll (void** ppvMemHeap) {
   /* should remove all allocated memory. there is no analog in standard memory
      management. */
}

/* Frees all memory and heap structure as well (if was allocated) */
```

```
void rtMemHeapRelease (void** ppvMemHeap) {
   /* should free all memory allocated + free memory heap object if exists */
}
```

In most cases it is only necessary to implement the following functions: *rtMemHeapAlloc*, *rtMemHeapAllocZ*, *rtMemHeapFreePtr* and *rtMemHeapRealloc*. Note that there is no analog in standard memory management for ASN1C's *rtMemFree* macro (i.e. the *rtMemHeapFreeAll* function). A user will need to be responsible for freeing all items in a generated ASN1C structure individually if standard memory management is used.

The *rtMemHeapCreate* and *rtMemHeapRelease* functions are specialized functions used when a special heap is to be used for allocation (for example, a static block within an embedded system). In this case, *rtMemHeapCreate* must set the *ppvMemHeap* argument to point at the block of memory to be used. This will then be passed in to all of the other memory management functions for their use through the ASN1CTXT structure. The *rtMemHeapRelease* function can then be used to dispose of this memory when it is no longer needed.

To add these definitions to an application program, compile the C source file (it can have any name) and link the resulting object file (.OBJ or .O) in with the application.

Low Level Memory Management API

It is possible to replace the core memory management functions used by the ASN1C run-time memory manager. This has the advantage of preserving the existing management scheme but with the use of different core functions. One reason for doing this might be an embedded system that does not have the standard C run-time functions *malloc*, *free*, and *realloc*.

To replace the core functions, the following global variables must be defined in a standard C file:

```
#include <stdlib.h>
#include "rtMemory.h"

ASN1UINT      g_defBlkSize = XM_K_MEMBLKSIZ;
OSMallocFunc  g_malloc_func = malloc;
OSReallocFunc g_realloc_func = realloc;
OSFreeFunc    g_free_func = free;
```

The *malloc, realloc,* and *free* functions must have the same prototype as the standard C functions.

The *g_defBlkSize* variable contains the default memory heap chunk size. This is the amount of memory allocated when the system needs a new block of memory. The default value is 4K (4096) bytes. The value can be set lower for space constrained systems and higher to improve performance in systems that have sufficient memory resources.

Some systems do not have a realloc-like function. In this case it is necessary to set the *g_realloc_func* variable to NULL. In this case the pair of *malloc/free* will be used (i.e. *g_malloc_func/ g_free_func* pair).

To add these definitions to an application program, compile the C source file (it can have any name) and link the resulting object file (.OBJ or .O) in with the application.

C++ Memory Management

In the case of C++, the ownership of memory is handled by the control class and message buffer objects. These classes share a context structure and use reference counting to manage the allocation and release of the context block. When a

message buffer object is created, a context block structure is created as well. When this object is then passed into a control class constructor, its reference count is incremented. Then when either the control class object or message buffer object are deleted or go out of scope, the count is decremented. When the count goes to zero (i.e. when both the message buffer object and control class object go away) the context structure is released.

What this means to the user is that a control class or message buffer object must be kept in scope when using a data structure associated with that class. A common mistake is to try and pass a data variable out of a method and use it after the control and message buffer objects go out of scope. For example, consider the following code fragment:

```
ASN1T_<type>* func2 () {
   ASN1T_<type>* p = new ASN1T_<type> ();
   ASN1BERDecodeBuffer decbuf;
   ASN1C_<type> cc (decbuf, *p);

   cc.Decode();

   // After return, cc and decbuf go out of scope; therefore
   // all memory allocated within struct p is released..

   return p;
}

void func1 () {
   ASN1T_<type>* pType = func2 ();

   // pType is not usable at this point because dynamic memory
   // has been released..

}
```

As can be seen from this example, once func2 exits, all memory that was allocated by the decode function will be released. Therefore, any items that require dynamic memory within the data variable will be in an undefined state.

An exception to this rule occurs when the type of the message being decoded is a *Protocol Data Unit* (PDU). These are the main message types in a specification. The ASN1C compiler designates types that are not referenced by any other types as PDU types. This behavior can be overridden by using the -pdu command line argument or <isPDU> configuration file element.

The significance of PDU types is that generated classes for these types are derived from the *ASN1TPDU* base class. This class holds a reference to a context object. The context object is set by Decode and copy methods. Thus, even if control class and message buffer objects go out of scope, the memory will not be freed until destructor of *ASN1TPDU* inherited class is called. The example above will work correctly without any modifications in this case.

Another way to keep data is to make copy of the decoded object before leave a scope. A method called *newCopy* is also generated in the control class for these types which can be used to create a copy of the decoded object. This copy of the object will persist after the control class and message buffer objects are deleted. The returned object can be deleted using the standard C++ *delete* operator when it is no longer needed.

Returning to the example above, it can be made to work if the type being decoded is a PDU type by doing the following:

```
ASN1T_<type>* func2 () {
```

```
   ASN1T_<type> msgdata;
   ASN1BERDecodeBuffer decbuf;
   ASN1C_<type> cc (decbuf, msgdata);

   cc.Decode();

   // Use newCopy to return a copy of the decoded item..

   return cc.newCopy();
}
```

The copy returned must be explicitly deleted using the C++ *delete* operator.

Populating Generated Structure Variables for Encoding

Prior to calling a compiler generated encode function, a variable of the type generated by the compiler must be populated. This is normally a straightforward procedure – just plug in the values to be encoded into the defined fields. However, things get more complicated when more complex, constructed structures are involved. These structures frequently contain pointer types which means memory management issues must be dealt with.

There are three alternatives for managing memory for these types:

1. Allocate the variables on the stack and plug the address of the variables into the pointer fields,
2. Use the standard **malloc** and **free** C functions to allocate memory to hold the data, and
3. Use the **rtMemAlloc** and **rtMemFree** run-time library functions or their associated macros.

Allocating the variables on the stack is an easy way to get temporary memory and have it released when it is no longer being used. But one has to be careful when using additional functions to populate these types of variables. A common mistake is the storage of the addresses of automatic variables in the pointer fields of a passed-in structure. An example of this error is as follows (assume A, B, and C are other structured types):

```
typedef struct {
   A* a;
   B* b;
   C* c;
} Parent;

void fillParent (Parent* parent)
{
   A aa;
   B bb;
   C cc;

   /* logic to populate aa, bb, and cc */
   ...

   parent->a = &aa;
   parent->b = &bb;
   parent->c = &cc;
}
```

```
main ()
{
   Parent parent;

   fillParent (&parent);

   encodeParent (&parent);   /*error: pointers in parent
   reference memory that is
   out of scope */
   ...
}
```

In this example, the automatic variables aa, bb, and cc go out of scope when the *fillParent* function exits. Yet the parent structure is still holding pointers to the now out of scope variables (this type of error is commonly known as "dangling pointers").

Using the second technique (i.e., using C *malloc* and *free*) can solve this problem. In this case, the memory for each of the elements can be safely freed after the encode function is called. But the downside is that a *free* call must be made for each corresponding *malloc* call. For complex structures, remembering to do this can be difficult thus leading to problems with memory leaks.

The third technique uses the compiler run-time library memory management functions to allocate and free the memory. The main advantage of this technique as opposed to using C *malloc* and *free* is that all allocated memory can be freed with a single *rtMemFree* call. The *ASN1MALLOC* macro can be used to allocate memory in much the same way as the C *malloc* function with the only difference being that a pointer to an initialized ASN1CTXT structure is passed in addition to the number of bytes to allocate. All allocated memory is tracked within the context structure so that when the *rtMemFree* function is called, all memory is released at once.

Accessing Encoded Message Components

After a message has been encoded, the user must obtain the start address and length of the message in order to do further operations with it. Before a message can be encoded, the user must describe the buffer the message is to be encoded into by specifying a message buffer start address and size. There are three different types of message buffers that can be described:

1. **static**: this is a fixed-size byte array into which the message is encoded

2. **dynamic**: in this case, the encoder manages the allocation of memory to hold the encoded message

3. **stream**: in this case, the encoder writes then encoded data directly to an output stream

The static buffer case is generally the better performing case because no dynamic memory allocations are required. However, the user must know in advance the amount of memory that will be required to hold an encoded message. There is no fixed formula to determine this number. ASN.1 encoding involves the possible additions of tags and lengths and other decorations to the provided data that will increase the size beyond the initial size of the populated data structures. The way to find out is either by trial-and-error (an error will be signaled if the provided buffer is not large enough) or by using a very large buffer in comparison to the size of the data.

In the dynamic case, the buffer description passed into the encoder is a null buffer pointer and zero size. This tells the encoder that it is to allocate memory for the message. It does this by allocating an initial amount of memory and when this is used up, it expands the buffer by reallocating. This can be an expensive operation in terms of performance –

especially if a large number of reallocations are required. For this reason, run-time helper functions are provided that allow the user to control the size increment of buffer expansions. See the *C/C++ Run-Time Library Reference Manual* for a description of these functions.

In either case, after a message is encoded, it is necessary to get the start address and length of the message. Even in the static buffer case, the message start address may be different then the buffer start address (see the section on encoding BER messages). For this reason, each set of encoding rules has a run-time C function for getting the message start address and length. See the *C/C++ Run-Time Library Reference Manual* for a description of these functions. The C++ message buffer classes contain the *getMsgPtr*, *getMsgCopy*, and *getMsgLength* methods for this purpose.

A **stream** message buffer can be used for BER encoding. This type of buffer is used when the –stream option was used to generate the encode functions. See the section on BER stream encoding for a complete description on how to set up an output stream to receive encoded data.

Generated BER Encode Functions

For each ASN.1 production defined in the ASN.1 source file, a C encode function is generated. This function will convert a filled-in C variable of the given type into an encoded ASN.1 message.

If C++ code generation is specified, a control class is generated that contains an *Encode* method that wraps this function. This function is invoked through the class interface to convert a populated *msgData* attribute variable into an encoded ASN.1 message.

Generated C Function Format and Calling Parameters

The format of the name of each generated encode function is as follows:

```
asn1E_[<prefix>]<prodName>
```

where <prodName> is the name of the ASN.1 production for which the function is being generated and <prefix> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each encode function is as follows:

```
len = asn1E_<name> (ASN1CTXT* pctxt,
                    <name>* pvalue,
                    ASN1TagType tagging);
```

In this definition, <name> denotes the prefixed production name defined above.

The pctxt argument is used to hold a context pointer to keep track of encode parameters. This is a basic "handle" variable that is used to make the function reentrant so it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her program. The variable should be initialized using the *rtInitContext* run-time library function (see the *C/C++ Common Run-Time Library Reference Manual* for a complete description of this function).

The pvalue argument holds a pointer to the data to be encoded and is of the type generated from the ASN.1 production.

The tagging argument is for internal use when calls to encode functions are nested to accomplish encoding of complex variables. It indicates whether the tag associated with the production should be applied or not (implicit versus explicit tagging). At the top level, the tag should always be applied so this parameter should always be set to the constant ASN1EXPL (for EXPLICIT).

The function result variable len returns the length of the data actually encoded or an error status code if encoding fails. Error status codes are negative to tell them apart from length values. Return status values are defined in the "asn1type.h" include file.

Generated C++ Encode Method Format and Calling Parameters

The C++ version of the compiler generates an *Encode* method that wraps the C function call. This method provides a more simplified calling interface because it hides things such as the context structure and the tag type parameters.

The calling sequence for the generated C++ class method is as follows:

```
len = <object>.Encode ();
```

In this definition, <object> is an instance of the control class (i.e., ASN1C_<prodName>) generated for the given production. The function result variable `len` returns the length of the data actually encoded or an error status code if encoding fails. Error status codes are negative to tell them apart from length values. Return status values are defined in the *asn1type.h* include file.

Procedure for Calling C Encode Functions

This section describes the step-by-step procedure for calling a C BER or DER encode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

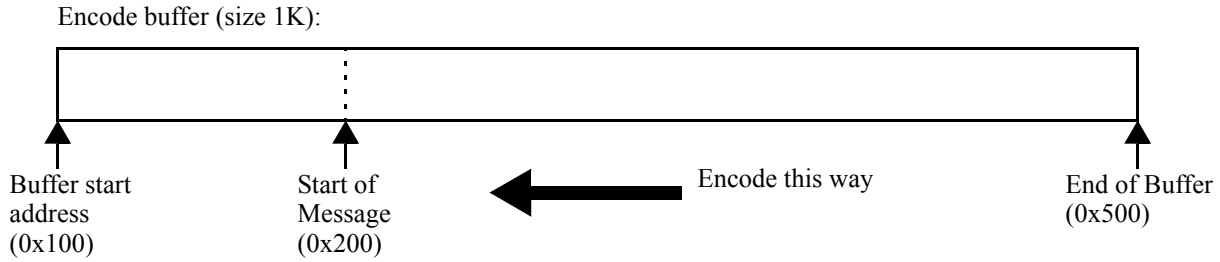
Before any encode function can be called; the user must first initialize an encoding context. This is a variable of type ASN1CTXT. This variable holds all of the working data used during the encoding of a message. The context variable is declared as a normal automatic variable within the top-level calling function. **It must be initialized before use**. This can be accomplished by using the *rtInitContext* function as follows:

```
ASN1CTXT ctxt;// context variable

if (rtInitContext (&ctxt) != ASN_OK) {
   /* initialization failed, could be a license problem */
   printf ("context initialization failed (check license)\n");
   return -1;
}
```

The next step is to specify an encode buffer into which the message will be encoded. This is accomplished by calling the *xe_setp* run-time function. The user can either pass the address of a buffer and size allocated in his or her program (referred to as a static buffer), or set these parameters to zero and let the encode function manage the buffer memory allocation (referred to as a dynamic buffer). Better performance can normally be attained by using a static buffer because this eliminates the high-overhead operation of allocating and reallocating memory.

After initializing the context and populating a variable of the structure to be encoded, an encode function can be called to encode the message. If the return status indicates success (positive length value), the run-time library function *xe_getp* can be called to obtain the start address of the encoded message. Note that the returned address is not the start address of the target buffer. BER encoded messages are constructed from back to front (i.e., starting at the end of the buffer and working backwards) so the start point will fall somewhere in the middle of the buffer after encoding is complete. This illustrated in the following diagram:



In this example, a 1K encode buffer is declared which happens to start at address 0x100. When the context is initialized with a pointer to this buffer and size equal to 1K, it positions the internal encode pointer to the end of the buffer (address 0x500). Encoding then proceeds from back-to-front until encoding of the message is complete. In this case, the encoded message turned out to be 0x300 (768) bytes in length and the start address fell at 0x200. This is the value that would be returned by the *xe_getp* function.

A program fragment that could be used to encode an employee record is as follows:

```
#include employee.h         /* include file generated by ASN1C */

int main ()
{
    ASN1OCTET msgbuf[1024], *msgptr;
    int       msglen;
    ASN1CTXT  ctxt;
    Employee  employee;/* typedef generated by ASN1C */

    /* Step 1: Initialize the context and set the buffer pointer */

    if (rtInitContext (&ctxt) != ASN_OK) {
       /* initialization failed, could be a license problem */
       printf ("context initialization failed (check license)\n");
       return -1;
    }

    xe_setp (&ctxt, msgbuf, sizeof(msgbuf));

    /* Step 2: Populate the structure to be encoded */

    employee.name.numocts = 5;
    employee.name.data = "SMITH";
    ...

    /* Step 3: Call the generated encode function */

    msglen = asn1E_Employee (&ctxt, &employee, ASN1EXPL);

    /* Step 4: Check the return status (note: the test is */
    /* > 0 because the returned value is the length of the */
```

```
    /* encoded message component)..*/

    if (msglen > 0) {

      /* Step 5: If encoding is successful, call xe_getp to */
      /* fetch a pointer to the start of the encoded message.*/

      msgptr = xe_getp (&ctxt);
      ...
    }
    else
      error processing...
}
```

In general, static buffers should be used for encoding messages where possible as they offer a substantial performance benefit over dynamic buffer allocation. The problem with static buffers, however, is that you are required to estimate in advance the approximate size of the messages you will be encoding. There is no built-in formula to do this, the size of an ASN.1 message can vary widely based on data types and the number of tags required.

If performance is not a significant an issue, then dynamic buffer allocation is a good alternative. Setting the buffer pointer argument to NULL in the call to *xe_setp* specifies dynamic allocation. This tells the encoding functions to allocate a buffer dynamically. The address of the start of the message is obtained as before by calling *xe_getp*. Note that this is not the start of the allocated memory; that is maintained within the context structure. To free the memory, the *rtMemFree* run-time function or *ASN1MEMFREE* macro must be called.

The following code fragment illustrates encoding using a dynamic buffer:

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
    ASN1OCTET *msgptr;
    int       msglen;
    ASN1CTXT  ctxt;
    Employee  employee;/* typedef generated by ASN1C */

    if (rtInitContext (&ctxt) != ASN_OK) {
       /* initialization failed, could be a license problem */
       printf ("context initialization failed (check license)\n");
       return -1;
    }

    xe_setp (&ctxt, NULL, 0);

    employee.name.numocts = 5;
    employee.name.data = "SMITH";
    ...

    msglen = asn1E_Employee (&ctxt, &employee, ASN1EXPL);

    if (msglen > 0) {
```

```
        msgptr = xe_getp (&ctxt);
        ...

        ASN1MEMFREE (&ctxt);   /* don't call free (msgptr); !!! */
      }
      else
        error processing...
  }
```

Encoding a Series of Messages Using the C Encode Functions

A common application of BER encoding is the repetitive encoding of a series of the same type of message over and over again. For example, a TAP3 batch application might read billing data out of a database table and encode each of the records for a batch transmission.

If a user was to repeatedly allocate/free memory and reinitialize the C objects involved in the encoding of a message, performance would suffer. This is not necessary however, because the C objects and memory heap can be reused to allow multiple messages to be encoded. As example showing how to do this is as follows:

```
#include employee.h         // include file generated by ASN1C

main ()
{
    const ASN1OCTET* msgptr;
    ASN1OCTET msgbuf[1024];
    int       msglen;
    ASN1CTXT  ctxt;
    PersonnelRecord data;

    /* Init context structure */

    if ((stat = rtInitContext (&ctxt)) != ASN_OK) {
        printf ("rtInitContext failed; stat = %d\n", stat);
        return -1;
    }

    // Encode loop starts here, this will repeatedly use the
    // objects declared above to encode the messages

    for (;;) {

        xe_setp (&ctxt, msgbuf, sizeof(msgbuf));

        // logic here to read record from some source (database,
        // flat file, socket, etc.)..

        // populate structure with data to be encoded

        data.name = "SMITH";
        ...
```

```
            // invoke Encode method

            if ((msglen = asn1E_PersonnelRecord (&ctxt, &data, ASN1EXPL)) > 0) {

                // encoding successful, get pointer to start of message

                msgptr = xe_getp (&ctxt);

                // do something with the encoded message

                ...
            }
            else
               error processing...

            // Call the rtMemReset to reset the memory heap for the next
            // iteration. Note, all data allocated by rtMemAlloc will become
            // invalid after this call.
            rtMemReset (&ctxt);
        }

        rtFreeContext (&ctxt);
    }
```

Note, the rtMemReset actually does not free the memory heap - it just marks it as empty and it will be reused in the next iteration. Thus, all allocated by rtMemAlloc memory will be overwritten and data will be lost. If user really needs to keep some data allocated by rtMemAlloc through all iterations you need either to copy it or use another context's instance to allocate memory for such data.

Procedure for Using the C++ Control Class Encode Method

The procedure to encode a message using the C++ class interface is as follows:

1. Create a variable of the *ASN1T_<name>* type and populate it with the data to be encoded.
2. Create an *ASN1BEREncodeBuffer* object.
3. Create a variable of the generated *ASN1C_<name>* class specifying the items created in 1 and 2 as arguments to the constructor.
4. Invoke the *Encode* method.

The constructor of the *ASN1C_<type>* class takes a message buffer object argument. This makes it possible to specify a static encode message buffer when the class variable is declared. A static buffer can improve encoding performance greatly as it relieves the internal software from having to repeatedly resize the buffer to hold the encoded message. If you know the general size of the messages you will be sending, or have a fixed size maximum message length, then a static buffer should be used. The message buffer argument can also be used to specify the start address and length of a received message to be decoded.

After the data to be encoded is set, the *Encode* method is called. This method returns the length of the encoded message or a negative value indicating that an error occurred. The error codes can be found in the asn1type.h run-time header file or in Appendix A of the C/C++ Common Functions Reference Manual of this document.

If encoding is successful, a pointer to the encoded message can be obtained by using the *getMsgPtr* or *getMsgCopy* methods available in the *ASN1BEREncodeBuffer* class. The *getMsgPtr* method is faster as it simply returns a pointer to

the actual start-of-message that is maintained within the message buffer object. The *getMsgCopy* method will return a copy of the message. Memory for this copy will be allocated using the standard new operator, so it up to the user to free this memory using delete when finished with the copy.

A program fragment that could be used to encode an employee record is as follows. This example uses a static encode buffer:

```
#include employee.h         // include file generated by ASN1C

main ()
{
    const ASN1OCTET* msgptr;
    ASN1OCTET msgbuf[1024];
    int       msglen;

    // step 1: construct ASN1C C++ generated class.
    // this specifies a static encode message buffer

    ASN1BEREncodeBuffer encodeBuffer (msgbuf, sizeof(msgbuf));
    ASN1T_PersonnelRecord msgData;
    ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

    // step 2: populate msgData structure with data to be encoded
    // (note: this uses the generated assignment operator to assign
    // a string)..

    msgData.name = "SMITH";
    ...


    // step 3: invoke Encode method

    if ((msglen = employee.Encode ()) > 0) {
       // encoding successful, get pointer to start of message
       msgptr = encodeBuffer.getMsgPtr();
    }
    else
      error processing...
}
```

The following code fragment illustrates encoding using a dynamic buffer. This also illustrates using the *getMsgCopy* method to fetch a copy of the encoded message:

```
#include employee.h         // include file generated by ASN1C

main ()
{
    ASN1OCTET* msgptr;
    int        msglen;
```

```
    // construct encodeBuffer class with no arguments

    ASN1BEREncodeBuffer encodeBuffer;
    ASN1T_PersonnelRecord msgData;
    ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

    // populate msgData structure

    msgData.name = "SMITH";
    ...

    // call Encode method

    if ((msglen = employee.Encode ()) > 0) {
      // encoding successful, get copy of message
      msgptr = encodeBuffer.getMsgCopy();
      ...

      delete [] msgptr;  // free the dynamic memory!
    }
    else
      error processing...
}
```

Encoding a Series of Messages Using the C++ Control Class Interface

A common application of BER encoding is the repetitive encoding of a series of the same type of message over and over again. For example, a TAP3 batch application might read billing data out of a database table and encode each of the records for a batch transmission.

If a user was to repeatedly instantiate and destroy the C++ objects involved in the encoding of a message, performance would suffer. This is not necessary however, because the C++ objects can be reused to allow multiple messages to be encoded. As example showing how to do this is as follows:

```
#include employee.h         // include file generated by ASN1C

main ()
{
    const ASN1OCTET* msgptr;
    ASN1OCTET msgbuf[1024];
    int       msglen;

    ASN1BEREncodeBuffer encodeBuffer (msgbuf, sizeof(msgbuf));
    ASN1T_PersonnelRecord msgData;
    ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

    // Encode loop starts here, this will repeatedly use the
    // objects declared above to encode the messages

    for (;;) {
```

```
        // logic here to read record from some source (database,
        // flat file, socket, etc.)..

        // populate structure with data to be encoded

        msgData.name = "SMITH";
        ...

        // invoke Encode method

        if ((msglen = employee.Encode ()) > 0) {

            // encoding successful, get pointer to start of message

            msgptr = encodeBuffer.getMsgPtr();

            // do something with the encoded message

            ...
        }
        else
           error processing...

        // Call the init method on the encodeBuffer object to
        // prepare the buffer for encoding another message..

        encodeBuffer.init();
    }
}
```

Generated BER Decode Functions

For each ASN.1 production defined in an ASN.1 source file, a C decode function is generated. This function will decode an ASN.1 message into a C variable of the given type.

If C++ code generation is specified, a control class is generated that contains a *Decode* method that wraps this function. This function is invoked through the class interface to decode an ASN.1 message into the variable referenced in the *msgData* component of the class.

Generated C Function Format and Calling Parameters

The format of the name of each decode function generated is as follows:

```
asn1D_[<prefix>]<prodName>
```

where <prodName> is the name of the ASN.1 production for which the function is being generated and <prefix> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each decode function is as follows:

```
status = asn1D_<name> (ASN1CTXT* pctxt,
                       <name> *pvalue,
                       ASN1TagType tagging,
                       int length);
```

In this definition, <name> denotes the prefixed production name defined above.

The `pctxt` argument is used to hold a context pointer to keep track of decode parameters. This is a basic "handle" variable that is used to make the function reentrant so it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her program. The variable must be initialized using the *rtInitContext* run-time function before use.

The `pvalue` argument is a pointer to a variable of the generated type that will receive the decoded data.

The `tagging` and `length` arguments are for internal use when calls to decode functions are nested to accomplish decoding of complex variables. At the top level, these parameters should always be set to the constants ASN1EXPL and zero respectively.

The function result variable `status` returns the status of the decode operation. The return status will be zero (ASN_OK) if decoding is successful or negative if an error occurs. Return status values are defined in the "asn1type.h" include file.

Generated C++ Decode Method Format and Calling Parameters

Generated decode functions are invoked through the class interface by calling the base class 'Decode' method. The calling sequence for this method is as follows:

```
status = <object>.Decode ();
```

In this definition, <object> is an instance of the control class (i.e., ASN1C_<prodName>) generated for the given production.

An *ASN1BERDecodeBuffer* object reference is a required argument to the <object> constructor. This is where the message start address and length are specified.

The message length argument is used to specify the size of the message, if it is known. In ASN.1 BER or DER encoded messages, the overall length of the message is embedded in the first few bytes of the message, so this variable is not required. It is used as a test mechanism to determine if a corrupt or partial message was received. If the parsed message length is greater than this value, an error is returned. If the value is specified to be zero (the default), then this test is bypassed.

The function result variable `status` returns the status of the decode operation. The return status will be zero (ASN_OK) if decoding is successful or a negative value if an error occurs. Return status values are defined in Appendix A of the C/C++ Common Functions Reference Manual of this document and online in the "asn1type.h" include file.

Procedure for Calling C Decode Functions

This section describes the step-by-step procedure for calling a C BER or DER decode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

Before any decode function can be called; the user must first initialize a context variable. This is a variable of type ASN1CTXT. This variable holds all of the working data used during the decoding of a message. The context variable is declared as a normal automatic variable within the top-level calling function. **It must be initialized before use**. This can be accomplished as follows:

```
ASN1CTXT ctxt;// context variable

if (rtInitContext (&ctxt) != ASN_OK) {
   /* initialization failed, could be a license problem */
   printf ("context initialization failed (check license)\n");
   return -1;
}
```

The next step is the specification of a buffer containing a message to be decoded. This is accomplished by calling the *xd_setp* run-time library function. This function takes as an argument the start address of the message to be decoded. The function returns the starting tag value and overall length of the message. This makes it possible to identify the type of message received and apply the appropriate decode function to decode it.

A decode function can then be called to decode the message. If the return status indicates success, the C variable that was passed as an argument will contain the decoded message contents. Note that the decoder may have allocated dynamic memory and stored pointers to objects in the C structure. After processing on the C structure is complete, the run-time library function *rtMemFree* or the macro *ASN1MEMFREE* should be called to free the allocated memory.

A program fragment that could be used to decode an employee record is as follows:

```
#include employee.h         /* include file generated by ASN1C */
```

```
main ()
{
    ASN1OCTET msgbuf[1024];
    ASN1TAG   msgtag;
    int       msglen;
    ASN1CTXT  ctxt;
    PersonnelRecord employee;

    .. logic to read message into msgbuf ..

    /* Step 1: Initialize a context variable for decoding */

    if (rtInitContext (&ctxt) != ASN_OK) {
       /* initialization failed, could be a license problem */
       printf ("context initialization failed (check license)\n");
       return -1;
    }

    xd_setp (&ctxt, msgbuf, 0, &msgtag, &msglen);

    /* Step 2: Test message tag for type of message received */
    /* (note: this is optional, the decode function can be */
    /* called directly if the type of message is known)..*/

    if (msgtag == TV_PersonnelRecord)
    {
        /* Step 3: Call decode function (note: last two args*/
        /* should always be ASN1EXPL and 0)..*/

        status = asn1D_PersonnelRecord (&ctxt,
                                        &employee,
                                        ASN1EXPL, 0);

        /* Step 4: Check return status */

        if (status == ASN_OK)
        {
            process received data in 'employee' variable..

            /* Remember to release dynamic memory when done! */

            ASN1MEMFREE (&ctxt);
        }
        else
            error processing...
    }
    else
       check for other known message types..
}
```

Decoding a Series of Messages Using the C Decode Functions

The above example is fine as a sample for decoding a single message, but what happens in the more typical scenario of having a long-running loop that continuously decodes messages? It will be necessary to put the decoding logic into a loop. A code fragment showing a way to do this is as follows:

```
main ()
{
    ASN1OCTET msgbuf[1024];
    ASN1TAG   msgtag;
    int       msglen;
    ASN1CTXT  ctxt;
    PersonnelRecord employee;

    /* Step 1: Initialize a context variable for decoding */

    if (rtInitContext (&ctxt) != ASN_OK) {
       /* initialization failed, could be a license problem */
       printf ("context initialization failed (check license)\n");
       return -1;
    }

    for (;;) {

       .. logic to read message into msgbuf ..

       xd_setp (&ctxt, msgbuf, 0, &msgtag, &msglen);

       /* Step 2: Test message tag for type of message received */
       /* (note: this is optional, the decode function can be */
       /* called directly if the type of message is known)..*/

       /* Now switch on initial tag value to determine what type of
          message was received.. */

       switch (msgtag)
       {
          case TV_PersonnelRecord:  // compiler generated constant
          {
             status = asn1D_PersonnelRecord (&ctxt,
                                             &employee,
                                             ASN1EXPL, 0);
             if (status == ASN_OK)
             {
                /* decoding successful, data in employee */

                process received data..
             }
             else
                error processing...
          }
```

```
            break;

            default:
               /* handle unknown message type here */

         }  /* switch */

         /* Need to reinitialize objects for next iteration */

         rtMemReset (&ctxt);
      }
   }
```

The only changes were the addition of the *for (;;)* loop and the call to *rtMemReset* that was added at the bottom of the loop. This function resets the memory tracking parameters within the context to allow previously allocated memory to be reused for the next decode operation. Optionally, *rtMemFree* can be called to release all memory. This will allow the loop to start again with no outstanding memory allocations for the next pass.

The example above assumes that logic existed that would read each message to be processed into the same buffer for every message processed inside the loop (i.e the buffer is reused each time). In the case in which the buffer already contains multiple messages, encoded back-to-back, it is necessary to advance the buffer pointer in each iteration:

```
main ()
{
    ASN1OCTET msgbuf[1024];
    ASN1TAG   msgtag;
    int       offset = 0, msglen, len;
    ASN1CTXT  ctxt;
    PersonnelRecord employee;
    FILE* fp;

    /* Step 1: Initialize a context variable for decoding */

    if (rtInitContext (&ctxt) != ASN_OK) {
       /* initialization failed, could be a license problem */
       printf ("context initialization failed (check license)\n");
       return -1;
    }

    if (fp = fopen (filename, "rb")) {
       msglen = fread (msgbuf, 1, sizeof(msgbuf), fp);
    }
    else {
       ... handle error ...
    }

    for (;offset < msglen; ) {
       xd_setp (&ctxt, msgbuf + offset, msglen - offset, &msgtag, &len);

       /* Decode */
```

```
         if (tag == TV_PersonnelRecord) {

            /* Call compiler generated decode function */

            stat = asn1D_PersonnelRecord (&ctxt, &employee, ASN1EXPL, 0);
            if (stat == ASN_OK) {

                /* decoding successful, data in employee */

            }
            else {

               /* error handling */
               return -1;
            }
         }
         else {
            printf ("unexpected tag %hx received\n", tag);
         }
         offset += ctxt.buffer.byteIndex;
         rtMemReset (&ctxt);
      }
   }
```

Procedure for Using the C++ Control Class Decode Method

Normally when a message is received and read into a buffer, it can be one of several different message types. So the first job a programmer has before calling a decode function is determining which function to call. The *ASN1BERDecodeBuffer* class has a standard method for parsing the initial tag/length from a message to determine the type of message received. This call is used in conjunction with a switch statement on generated tag constants for the known message set in order to pick a decoder to call.

Once it is known which type of message has been received, an instance of a generated message class can be instantiated and the decode function called. The start of message pointer and message length (if known) must be specified either in the constructor call or in the call to the decode function itself.

A program fragment that could be used to decode an employee record is as follows:

```
#include employee.h         // include file generated by ASN1C

main ()
{
    ASN1OCTET msgbuf[1024];
    ASN1TAG   msgtag;
    int       msglen, status;

    .. logic to read message into msgbuf ..

    // Use the ASN1BERDecodeBuffer class to parse the initial
    // tag/length from the message..
```

```
ASN1BERDecodeBuffer decodeBuffer (msgbuf, len);

status = decodeBuffer.ParseTagLen (msgtag, msglen);

if (status != ASN_OK) {
   // handle error
   ...
}

// Now switch on initial tag value to determine what type of
// message was received..

switch (msgtag)
{
   case TV_PersonnelRecord:  // compiler generated constant
   {
      ASN1T_PersonnelRecord msgData;
      ASN1C_PersonnelRecord employee (decodeBuffer, msgData);

      if ((status = employee.Decode ()) == ASN_OK)
      {
         // decoding successful, data in msgData

         process received data..
      }
      else
         error processing...
   }

   case TV_ ...// handle other known messages
```

Note that the call to free memory is not required to release dynamic memory when using the C++ interface. This is because the control class hides all of the details of managing the context and releasing dynamic memory. The memory is automatically released when both the message buffer object (ASN1BERMessageBuffer) and the control class object (ASN1C_<ProdName>) are deleted or go out of scope. Reference counting of a context variable shared by both interfaces is used to accomplish this.

Decoding a Series of Messages Using the C++ Control Class Interface

The above example is fine as a sample for decoding a single message, but what happens in the more typical scenario of having a long-running loop that continuously decodes messages? The logic shown above would not be optimal from a performance standpoint because of the constant creation and destruction of the message processing objects. It would be much better to create all of the required objects outside of the loop and then reuse them to decode and process each message.

A code fragment showing a way to do this is as follows:

```
#include employee.h         // include file generated by ASN1C

main ()
```

```
{
    ASN1OCTET msgbuf[1024];
    ASN1TAG   msgtag;
    int       msglen, status;

    // Create message buffer, ASN1T, and ASN1C objects

    ASN1BERDecodeBuffer decodeBuffer (msgbuf, len);
    ASN1T_PersonnelRecord employeeData;
    ASN1C_PersonnelRecord employee (decodeBuffer, employeeData);

    for (;;) {

      .. logic to read message into msgbuf ..

      status = decodeBuffer.ParseTagLen (msgtag, msglen);

      if (status != ASN_OK) {
        // handle error
        ...
      }

     // Now switch on initial tag value to determine what type of
     // message was received..

     switch (msgtag)
     {
       case TV_PersonnelRecord:  // compiler generated constant
       {
          if ((status = employee.Decode ()) == ASN_OK)
          {
             // decoding successful, data in employeeData

             process received data..
          }
          else
             error processing...
       }
       break;

       default:
          // handle unknown message type here

      }  // switch

      // Need to reinitialize objects for next iteration

      if (!isLastIteration) employee.memFreeAll ();

    }  // end of loop
```

This is quite similar to the first example. Note that we have pulled the *ASN1T_Employee* and *ASN1C_Employee* object creation logic out of the switch statement and moved it above the loop. These objects can now be reused to process each received message.

The only other change was the call to *employee.memFreeAll* that was added at the bottom of the loop. Since we can't count on the objects being deleted to automatically release allocated memory, we need to do it manually. This call will free all memory held within the decoding context. This will allow the loop to start again with no outstanding memory allocations for the next pass.

If buffer already contains multiple BER messages encoded back-to-back then it is necessary to modify buffer's pointer in each iteration. *decodeBuffer.getByteIndex ()* method should be used at the end of loop to get current offset in the buffer. This offset should be used for *decodeBuffer.setBuffer* method call at the beginning of the loop to determine correct buffer pointer and length:

```
ASN1UINT offset = 0;
for (;offset < msglen;) {

  // set buffer pointer and its length to decode

  decodeBuffer.setBuffer (&msgbuf[offset], msglen - offset);
  int curlen = (int)(msglen - offset);

  status = decodeBuffer.ParseTagLen (msgtag, curlen);

  if (status != ASN_OK) {
    // handle error
    ...
  }

  // Now switch on initial tag value to determine what type of
  // message was received..

  switch (msgtag)
  {
    case TV_PersonnelRecord:  // compiler generated constant
    {
      if ((status = employee.Decode ()) == ASN_OK)
      {
         // decoding successful, data in employeeData

         process received data..
      }
      else
         error processing...
    }
    break;

    default:
      // handle unknown message type here
```

```
      }  // switch

      // get new offset
      offset += decodeBuffer.getByteIndex ();

      // Need to reinitialize objects for next iteration (if it is not
      // last iteration)

      if (offset < msglen) employee.memFreeAll ();

    }  // end of loop
```

BER Decode Performance Enhancement Techniques

There are a number of different things that can be done in application code to improve BER decode performance. These include adjusting memory allocation parameters, using compact code generation, and using decode fast copy.

Dynamic Memory Management

By far, the biggest performance bottleneck when decoding ASN.1 messages is the allocation of memory from the heap. Each call to **new** or **malloc** is very expensive.

The decoding functions must allocate memory because the sizes of many of the variables that make up a message are not known at compile time. For example, an OCTET STRING that does not contain a size constraint can be an indeterminate number of bytes in length.

ASN1C does two things by default to reduce dynamic memory allocations and improve decoding performance:

1. Uses static variables wherever it can. Any BIT STRING, OCTET STRING, character string, or SEQUENCE OF or SET OF construct that contains a size constraint will result in the generation of a static array of elements sized to the max constraint bound.

2. Uses a special nibble-allocation algorithm for allocating dynamic memory. This algorithm allocates memory in large blocks and then splits up these blocks on subsequent memory allocation requests. This results in fewer calls to the kernel to get memory. The downside is that one request for a few bytes of memory can result in a large block being allocated.

Common run-time functions are available for controlling the memory allocation process. First, the default size of a memory block as allocated by the nibble-allocation algorithm can be changed. By default, this value is set to 4K bytes. The run-time function *rtMemSetDefBlkSize* can be called to change this size. This takes a single argument - the value to which the size should be changed.

It is also possible to change the underlying functions called from within the memory management abstraction layer to obtain or free heap memory. By default, the standard C *malloc*, *realloc*, and *free* functions are used. These can be changed by calling the *rtMemSetAllocFuncs* function. This function takes as arguments function pointers to the allocate, reallocate, and free functions to be used in place of the standard C functions.

Another run-time memory management function that can improve performance is *rtMemReset*. This function is useful when decoding messages in a loop. It is used instead of *rtMemFree* at the bottom of the loop to make dynamic memory available for decoding the next message. The difference is that *rtMemReset* does not actually free the dynamic memory.

It instead just resets the internal memory management parameters so that memory already allocated can be reused. Therefore, all the memory required to handle message decoding is normally allocated within the first few passes of the loop. From that point on, that memory is reused thereby making dynamic memory allocation a negligent issue in the overall performance of the decoder.

A more detailed explanation of these functions and other memory management functions can be found in the *C/C++Common Run-Time Library Reference Manual*.

Compact Code Generation

Using the compact code generation option (*-compact*) and lax validation option (*-lax*) can also improve decoding performance.

The *-compact* option causes code to be generated that contains no diagnostic or error trace messages. In addition, some status checks and other non-critical code are removed providing a slightly less robust but faster code base.

The *–lax* option causes all constraint checks to be removed from the generated code.

Performance intensive applications should also be sure to link with the compact version of the base run-time libraries. These libraries can be found in the *lib_opt* (for optimized) subdirectory. These run-time libraries also have all diagnostics and error trace messages removed as well as some non-critical status checks.

Decode Fast Copy

"Fast Copy" is a special run-time flag that can be set for the decoder that can substantially reduce the number of copy operations that need to be done to decode a message. The copy operations are reduced by taking advantage of the fact that the data contents of some ASN.1 types already exist in decoded form in the message buffer. Therefore, there is no need to allocate memory for the data and then copy the data from the buffer into the allocated memory structure.

As an example of what fast copy does, consider a simple ASN.1 SEQUENCE consisting of an element a, an INTEGER and b, an OCTET STRING:

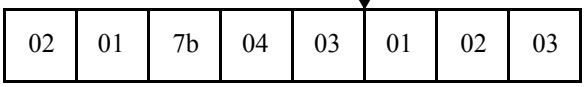
```
Simple ::= SEQUENCE {
   a  INTEGER,
   b  OCTET STRING
}
```

Assume an encoded value of this type contains a value of a = 123 (hex 7B) and b contains the hex octets 0x01 0x02 0x03. The generated variable for the OCTET STRING will contain a data pointer. So rather than allocate memory for this string and copy the data to it, fast copy will simply store a pointer directly to the data in the buffer:

```
Simple.a =123

Simple.b.numocts = 3
Simple.b.data = ptr
```

Message buffer:



The pointer stored in the data structure points directly at data in the message buffer. No memory allocation or copy is done.

The user must keep in mind that if this technique is used, the message buffer containing the decoded message must be available as long as the type variable containing the decoded data is in use. This will not work in a producer-consumer threading model where one thread is decoding messages and the next thread is processing the contents. The producer thread will overwrite the buffer contents and therefore data referenced in the decoded message type variable that the consumer is processing.

This will also not work if the message buffer is an automatic variable in a function and the decoded result type is being passed out. The result type variable will point at data in the buffer variable that has gone out of scope.

To set fast copy, the *rtSetFastCopy* function must be invoked with the initialized context variable that will be used to decode a message. This should be done once prior to entering the loop that will be used to decode a stream of messages.

BER/DER Deferred Decoding

Another way to improve decoding performance of large messages is through the use of *deferred decoding*. This allows for the selective decoding of only parts of a message in a single decode function call. When combined with the fast copy procedure defined above, this can significantly reduce decoding time because large parts of messages can be skipped.

Deferred decoding can be done on elements defined within a SEQUENCE, SET or CHOICE construct. It is done by designating an element to be an open type by using the <*isOpenType/*> configuration setting. This setting causes the ASN1C compiler to insert an *Asn1OpenType* placeholder in place of the type that would have normally been used for the element. The data in its original encoded form will be stored in the open type container when the message is decoded. If fast copy is used, only a pointer to the data in the message buffer is stored so large copies of data are avoided.

The data within the deferred decoding open type container can be fully decoded later by using a special decode function generated by the ASN1C compiler for this purpose. The format of this function is as follows:

```
asn1D_<ProdName>_<ElementName>_OpenType (ASN1CTXT* pctxt, <ElementType>* pvalue)
```

Here <*ProdName*> is replaced with name of the type assignment and <*ElementName* > is replaced with name of the element. In this function, the argument *pctxt* is used to pass the a pointer to a context variable initialized with the open type data and the *pvalue* argument will hold the final decoded data value.

In following example, decoding of the element *id* is deferred:

```
  Identifier ::= SEQUENCE {
     id INTEGER,
     oid OBJECT IDENTIFIER
  }
```

The following configuration file is required to indicate the element *id* is to be processed as an open type (i.e. that it will be decoded later):

```
<asn1config>
   <module>
      <name>modulename</name>
      <production>
         <name>Identifier</name>
         <element>
            <name>id</name>
            <isOpenType/>
         <element/>
      <production/>
   <module/>
<asn1config/>
```

In the generated code, the element *id* type will be replaced with an open type (*Asn1OpenType* for C or *Asn1TOpenType* for C++) and the following additional function is generated:

```
EXTERN int asn1D_Identifier_id_OpenType (ASN1CTXT* pctxt, ASN1INT* pvalue);
```

In the *Identifier* decode function, element *id* is decoded as an open type.

Generated BER Stream-Oriented Encode Functions

BER messages can be encoded directly to an output stream such as a file, network or memory stream. The ASN1C compiler has the **–stream** option to generate encode functions of this type. For each ASN.1 production defined in the ASN.1 source file, a C stream encode function is generated. This function will encode a populated C variable of the given type into an encoded ASN.1 message and write it to a stream.

If the return status indicates success (ASN_OK), the message will have been encoded to the given stream. Stream-oriented BER encoding starts from the beginning of the message until the message is complete. This is sometimes referred to as "forward encoding". This differs from regular BER where encoding is done from back-to-front. Indefinite lengths are used for all constructed elements in the message. Also, there is no permanent buffer for stream-oriented encoding, all octets are written to the stream. The buffer in the context structure is used only as a cache.

If C++ code generation is specified, a control class is generated that contains an *EncodeTo* method that wraps the stream encode C function. This function is invoked through the class interface to convert a populated *msgData* attribute variable into an encoded ASN.1 message.

Generated Stream-Oriented C Function Format and Calling Parameters

The format of the name of each generated stream-oriented encode function is as follows:

```
asn1BSE_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each encode function is as follows:

```
stat = asn1BSE_<name> (ASN1CTXT*     pctxt,
                       <name>*     pvalue,
                       ASN1TagType tagging);
```

In this definition, <name> denotes the prefixed production name defined above.

The `pctxt` argument is used to hold a context pointer to keep track of encode parameters. This is a basic "handle" variable that is used to make the function reentrant so it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her program. This variable must be initialized using both the *rtInitContext* and *rtStreamBufInit* run-time library functions (see the *C/C++ Common Run-Time Library Reference Manual* for a description of these functions).

The `pvalue` argument holds a pointer to the data to be encoded and is of the type generated from the ASN.1 production.

The `tagging` argument is for internal use when calls to encode functions are nested to accomplish encoding of complex variables. It indicates whether the tag associated with the production should be applied or not (implicit versus explicit tagging). At the top level, the tag should always be applied so this parameter should always be set to the constant ASN1EXPL (for EXPLICIT).

The function result variable `stat` returns the completion status of the operation. 0 (ASN_OK) means the success.

Procedure for Calling Stream-Oriented C Encode Functions

This section describes the step-by-step procedure for calling a stream-oriented C BER encode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

Before any encode function can be called; the user must first initialize an encoding context. This is a variable of type ASN1CTXT. This variable holds all of the working data used during the encoding of a message. The context variable is within the top-level calling function. **It must be initialized before use**. This can be accomplished by using the *rtInitContext* function. The context also must be initialized for streaming operations by calling the *rtStreamBufInit* function:

```
ASN1CTXT ctxt;// context variable

if (rtInitContext (&ctxt) != ASN_OK) {
   /* initialization failed, could be a license problem */
   printf ("context initialization failed (check license)\n");
   return -1;
}

rtStreamBufInit (&ctxt);
```

The next step is to create a stream object within the context. This object is an abstraction of the output device to which the data is to be encoded and is initialized by calling one of the following functions:

- *rtStreamFileOpen*
- *rtStreamFileAttach*
- *rtStreamSocketAttach*
- *rtStreamMemoryCreate*
- *rtStreamMemoryAttach*

The *flags* parameter of these functions should be set to the OSRTSTRMF_OUTPUT constant value to indicate an output stream is being created (see the *C/C++ Common Run-Time Library Reference Manual* for a full description of these functions).

After initializing the context and populating a variable of the structure to be encoded, an encode function can be called to encode the message to the stream. The stream must then be closed by calling the *rtStreamBufClose* function.

A program fragment that could be used to encode an employee record is as follows:

```
#include employee.h         /* include file generated by ASN1C */

int main ()
{
    int       stat;
    ASN1CTXT  ctxt;
    Employee  employee;/* typedef generated by ASN1C */
    const char* filename = "message.dat";

    /* Step 1: Initialize the context and stream */

    if (rtInitContext (&ctxt) != ASN_OK) {
```

```
      /* initialization failed, could be a license problem */
      printf ("context initialization failed (check license)\n");
      return -1;
   }

   rtStreamBufInit (&ctxt);

   /* Step 2: create a file stream object within the context */

   stat = rtStreamFileOpen (&ctxt, filename, OSRTSTRMF_OUTPUT);
   if (stat != ASN_OK) {
   xu_perror (&ctxt);
   return stat;
   }
   /* Step 3: Populate the structure to be encoded */

   employee.name.numocts = 5;
   employee.name.data = "SMITH";
   ...

   /* Step 4: Call the generated encode function */

   stat = asn1BSE_Employee (&ctxt, &employee, ASN1EXPL);

   /* Step 5: Check the return status and close the stream*/

   if (stat != ASN_OK) {

     ...error processing...

   }

   rtStreamBufClose (&ctxt);
}
```

In general, stream-oriented encoding is slower than regular (so called "buffer-oriented") encoding. However, in the case of stream-oriented encoding, it is not necessary to implement code to write or send the encoded data to an output device. The stream-oriented functions also use less memory because there is no need for a large destination memory buffer. For this reason, the final performance of the stream-oriented functions may be the same or better than "buffer-oriented" functions.

Encoding a Series of Messages Using the Stream-Oriented C Encode Functions

A common application of BER encoding is the repetitive encoding of a series of the same type of message over and over again. For example, a TAP3 batch application might read billing data out of a database table and encode each of the records for a batch transmission.

Encoding a series of messages using the stream-oriented C encode functions is very similar to encoding of one message. All that is necessary is to set up a loop in which the asn1BSE_<name> functions will be called. It is also possible to call different asn1BSE_<name> functions one after another. An example showing how to do this is as follows:

```
#include employee.h         // include file generated by ASN1C

int main ()
{
    int       stat;
    ASN1CTXT  ctxt;
    Employee  employee;/* typedef generated by ASN1C */
    const char* filename = "message.dat";

    /* Step 1: Initialize the context and stream */

    if (rtInitContext (&ctxt) != ASN_OK) {
       /* initialization failed, could be a license problem */
       printf ("context initialization failed (check license)\n");
       return -1;
    }

    rtStreamBufInit (&ctxt);
    stat = rtStreamFileOpen (&ctxt, filename, OSRTSTRMF_OUTPUT);
    if (stat != ASN_OK) {
       xu_perror (&ctxt);
       return stat;
    }

    for (;;) {
      /* Step 2: Populate the structure to be encoded */

      employee.name.numocts = 5;
      employee.name.data = "SMITH";
      ...

      /* Step 3: Call the generated encode function */

      stat = asn1BSE_Employee (&ctxt, &employee, ASN1EXPL);

      /* Step 4: Check the return status and break the loop
          if error occurs */

      if (stat != ASN_OK) {

        ...error processing...

        break;
      }
   }

   /* Step 5: Close the stream */

   rtStreamBufClose (&ctxt);
}
```

Generated Stream-Oriented C++ Encode Method Format and Calling Parameters

C++ code generation of stream-based encoders is selected by using the **–c++** and **–stream** compiler command line options. In this case, ASN1C generates an *EncodeTo* method that wraps the C function call. This method provides a more simplified calling interface because it hides things such as the context structure and tag type parameters.

The calling sequence for the generated C++ class method is as follows:

```
stat = <object>.EncodeTo (<outputStream>);
```

In this definition, <object> is an instance of the control class (i.e., ASN1C_<prodName>) generated for the given production.

The <outputStream> placeholder represents an output stream object type. This is an object derived from an *ASN1OutputStream* class.

The function result variable `stat` returns the completion status. Error status codes are negative. Return status values are defined in the *asn1type.h* include file.

Another way to encode a message using the C++ classes is to use the << streaming operator:

```
<outputStream> << <object>;
```

In this case, the exception *OSCStreamException* could be thrown if an error occurs.

Also, the method *Encode* without parameters is supported for backward compatibility. In this case it is necessary to create control class (i.e., ASN1C_<prodName>) using an output stream reference as the first parameter and *msgdata* reference as the second parameter of the constructor.

Procedure for Using the Stream-Oriented C++ Control Class Encode Method

The procedure to stream-oriented encode a message using the C++ class interface is as follows:

1. Create an *ASN1BEROutputStream* inherited object.
2. Create a variable of the *ASN1T_<name>* type and populate it with the data to be encoded.
3. Create a variable of the generated *ASN1C_<name>* class specifying the item created in 2 as an argument to the constructor.
4. Invoke the *EncodeTo* method or << operator.

A program fragment that could be used to encode an employee record is as follows. This example uses a file output stream:

```
#include employee.h         // include file generated by ASN1C

main ()
{
   int msglen;
   const char* filename = "message.dat"

   // step 1: construct output stream object.

   ASN1BERFileOutputStream out (filename);
   if (out.getStatus () != ASN_OK) {
      out.printErrorInfo ();
      return -1;
   }

   // step 2: construct ASN1C C++ generated class.

   ASN1T_PersonnelRecord msgData;
   ASN1C_PersonnelRecord employee (msgData);

   // step 3: populate msgData structure with data to be
   // encoded. (note: this uses the generated assignment
   // operator to assign a string).

   msgData.name = "SMITH";
   ...
   // step 4: invoke << operator or EncodeTo method

   out << employee;
   // or employee.EncodeTo (out); can be used here.
   if (out.getStatus () != ASN_OK) {
      printf ("Encoding failed. Status = %i\n", out.getStatus());
      out.printErrorInfo ();
      return -1;
   }

   if (trace) {
      printf ("Encoding was successful\n");
   }
}
```

Encoding a Series of Messages Using the Stream-Oriented C++ Control Class Interface

Encoding a series of messages using the stream-oriented C++ control class is similar to the C method of encoding. All that is necessary is to create a loop in which *EncodeTo* or *Encode* methods will be called (or the overloaded << streaming operator). It is also possible to call different *EncodeTo* methods (or *Encode* or operator <<) one after another. An example showing how to do this is as follows:

```
#include employee.h         // include file generated by ASN1C

int main ()
{
   const ASN1OCTET* msgptr;
   ASN1OCTET msgbuf[1024];
   int       msglen;
   const char* filename = "message.dat"

   // step 1: construct stream object.

   ASN1BERFileOutputStream out (filename);
   if (out.getStatus () != ASN_OK) {
      out.printErrorInfo ();
      return -1;
   }

   // step 2: construct ASN1C C++ generated class.

   ASN1T_PersonnelRecord msgData;
   ASN1C_PersonnelRecord employee (msgData);

   for (;;) {
      // step 3: populate msgData structure with data to be
      // encoded. (note: this uses the generated assignment
      // operator to assign a string).

      msgData.name = "SMITH";
      ...

      // step 4: invoke << operator or EncodeTo method

      out << employee;
      // or employee.EncodeTo (out); can be used here.
      if (out.getStatus () != ASN_OK) {
         printf ("Encoding failed. Status = %i\n", out.getStatus());
         out.printErrorInfo ();
         return -1;
      }

      if (trace) {
         printf ("Encoding was successful\n");
      }
   }
}
```

Generated BER Stream-Oriented Decode Functions

BER messages can be directly read and decoded from an input stream such as a file, network or memory stream using BER stream-oriented decode functions. The ASN1C compiler **–stream** option is used to generate decoders of this type. For each ASN.1 production defined in an ASN.1 source file, a C stream-oriented decode function is generated. This function will decode an ASN.1 message into a C variable of the given type.

If C++ code generation is specified, a control class is generated that contains a *DecodeFrom* method that wraps this function. This function is invoked through the class interface to decode an ASN.1 message into the variable referenced in the *msgData* component of the class.

In this version, there are three types of streams: file, socket and memory. The most useful are file and socket streams. It is possible to decode data directly from a file or socket without intermediate copying into memory. If the full amount of data is not available for reading then the behavior of these streams will be different: the file and memory input streams will report an error, the socket input stream will block until data is available or an I/O error occurs (for example, the remote side closes the connection).

Generated Stream-Oriented C Function Format and Calling Parameters

The format of the name of each stream-oriented decode function generated is as follows:

```
asn1BSD_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each decode function is as follows:

```
status = asn1BSD_<name> (ASN1CTXT* pctxt,
                         <name> *pvalue,
                         ASN1TagType tagging,
                         int length);
```

In this definition, <name> denotes the prefixed production name defined above.

The `pctxt` argument is used to hold a context pointer to keep track of decode parameters. This is a basic "handle" variable that is used to make the function reentrant so it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her program. The variable must be initialized using the *rtInitContext* run-time function before use.

To perform streaming operations, the context pointer `pctxt` must be also initialized as a stream by using the *rtStreamBufInit* run-time library function (see the *C/C++ Common Run-Time Library Reference Manual* for a description of this function).

The `pvalue` argument is a pointer to a variable of the generated type that will receive the decoded data.

The `tagging` and `length` arguments are for internal use when calls to decode functions are nested to accomplish decoding of complex variables. At the top level, these parameters should always be set to the constants ASN1EXPL and zero respectively.

The function result variable `status` returns the status of the decode operation. The return status will be zero (ASN_OK) if decoding is successful or negative if an error occurs. Return status values are defined in the "asn1type.h" include file.

Procedure for Calling Stream-Oriented C Decode Functions

This section describes the step-by-step procedure for calling a stream-oriented C BER decode function. This procedure must be followed if C code generation was done. This procedure can also be used as an alternative to using the control class interface if C++ code generation was done.

Before any decode function can be called; the user must first initialize a context variable. This is a variable of type ASN1CTXT. This variable holds all of the working data used during the decoding of a message. The context variable is declared as a normal automatic variable within the top-level calling function. **It must be initialized before use**. This can be accomplished by using the *rtInitContext* function. Also, the context must be initialized for streaming operations by calling the *rtStreamBufInit* function:

```
ASN1CTXT ctxt;// context variable

if (rtInitContext (&ctxt) != ASN_OK) {
   /* initialization failed, could be a license problem */
   printf ("context initialization failed (check license)\n");
   return -1;
}

rtStreamBufInit (&ctxt));// Initialize stream
```

The next step is to create a stream object within the context. This object is an abstraction of the output device to which the data is to be encoded and is initialized by calling one of the following functions:

- *rtStreamFileOpen*
- *rtStreamFileAttach*
- *rtStreamSocketAttach*
- *rtStreamMemoryCreate*
- *rtStreamMemoryAttach*

The *flags* parameter of these functions should be set to the OSRTSTRMF_INPUT constant value to indicate an input stream is being created (see the *C/C++ Common Run-Time Library Reference Manual* for a full description of these functions).

After initializing the context and populating a variable of the structure to be encoded, a decode function can be called to decode a message from the stream. If the return status indicates success, the C variable that was passed as an argument will contain the decoded message contents. Note that the decoder may have allocated dynamic memory and stored pointers to objects in the C structure. After processing on the C structure is complete, the run-time library function *rtMemFree* or the macro *ASN1MEMFREE* should be called to free the allocated memory.

After stream processing is complete, the stream is closed by invoking the *rtStreamBufClose* function. A program fragment that could be used to decode an employee record is as follows:

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
   ASN1TAG   msgtag;
   int       msglen;
   ASN1CTXT  ctxt;
   PersonnelRecord employee;
   const char* filename = "message.dat"

   /* Step 1: Initialize a context variable for decoding */

   if (rtInitContext (&ctxt) != ASN_OK) {
      /* initialization failed, could be a license problem */
      printf ("context initialization failed (check license)\n");
      return -1;
   }

   rtStreamBufInit (&ctxt);

   /* Step 2: Open the input stream to read data */

   stat = rtStreamFileOpen (&ctxt, filename, OSRTSTRMF_INPUT);

   if (stat != ASN_OK) {
      xu_perror (&ctxt);
      return stat;
   }

   /* Step 3: Test message tag for type of message received */
   /* (note: this is optional, the decode function can be */
   /* called directly if the type of message is known)..*/

   stat = rtStreamBufMark (&ctxt, 32);
   if (stat != ASN_OK) {
      xu_perror (&ctxt);
      return stat;
   }

   stat = berDecStrmTagAndLen (&ctxt, &msgtag, &msglen);
   if (stat != ASN_OK) {
      xu_perror (&ctxt);
      return stat;
   }

   if (msgtag == TV_PersonnelRecord)
   {
      stat = rtStreamBufReset (&ctxt);
```

```
      if (stat != ASN_OK) {
         xu_perror (&ctxt);
         return stat;
      }

      /* Step 4: Call decode function (note: last two args*/
      /* should always be ASN1EXPL and 0)..*/

      status = asn1BSD_PersonnelRecord (&ctxt,
                                        &employee,
                                        ASN1EXPL, 0);

     /* Step 5: Check return status */

      if (status == ASN_OK)
      {
         process received data in 'employee' variable..

      }
      else
         error processing...
   }
   else
        check for other known message types..

   /* Step 6: Close the stream */

   rtStreamBufClose (&ctxt);

   /* Remember to release dynamic memory when done! */

   rtFreeContext (&ctxt);
}
```

Decoding a Series of Messages Using the Stream-Oriented C Decode Functions

The above example is fine as a sample for decoding a single message, but what happens in the more typical scenario of having a long-running loop that continuously decodes messages? It will be necessary to put the decoding logic into a loop.

A code fragment showing a way to do this is as follows:

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
   ASN1TAG   msgtag;
   int       msglen, stat;
   ASN1CTXT  ctxt;
   PersonnelRecord employee;
   const char* filename = "message.dat"

   /* Step 1: Initialize a context variable for decoding */

   if (rtInitContext (&ctxt) != ASN_OK) {
      /* initialization failed, could be a license problem */
      printf ("context initialization failed (check license)\n");
      return -1;
   }

   rtStreamBufInit (&ctxt);

   /* Step 2: Open the input stream to read data */

   stat = rtStreamFileOpen (&ctxt, filename, OSRTSTRMF_INPUT);

   if (stat != ASN_OK) {
      xu_perror (&ctxt);
      return stat;
   }

   for (;;) {
      /* Step 3: Test message tag for type of message received */
      /* (note: this is optional, the decode function can be  */
      /* called directly if the type of message is known)..  */

      stat = rtStreamBufMark (&ctxt, 32);
      if (stat != ASN_OK) {
         xu_perror (&ctxt);
         return stat;
      }

      stat = berDecStrmTagAndLen (&ctxt, &msgtag, &msglen);
      if (stat != ASN_OK) {
         xu_perror (&ctxt);
         return stat;
      }

      if (msgtag == TV_PersonnelRecord)
      {
         stat = rtStreamBufReset (&ctxt);
```

```
            if (stat != ASN_OK) {
               xu_perror (&ctxt);
               return stat;
            }

            /* Step 4: Call decode function (note: last two args */
            /* should always be ASN1EXPL and 0).. */

            stat = asn1BSD_PersonnelRecord (&ctxt,
                                            &employee,
                                            ASN1EXPL, 0);

            /* Step 5: Check return status */

            if (stat == ASN_OK)
            {
               process received data in 'employee' variable..

            }
            else
               error processing...
         }
         else
         check for other known message types..

         /* Need to reset all memory for next iteration */

         rtMemReset (&ctxt);

      } /* end of loop */

      /* Step 6: Close the stream */

      rtStreamBufClose (&ctxt);

      /* Remember to release dynamic memory when done! */

      rtFreeContext (&ctxt);
   }
```

The only changes were the addition of the *for (;;)* loop and the call to *rtMemReset* that was added at the bottom of the loop. This function resets the memory tracking parameters within the context to allow previously allocated memory to be reused for the next decode operation. Optionally, *rtMemFree* can be called to release all memory. This will allow the loop to start again with no outstanding memory allocations for the next pass.

Generated Stream-Oriented C++ Decode Method Format and Calling Parameters

Generated C stream-oriented decode functions are invoked through the C++ class interface by calling the generated *DecodeFrom* method. The calling sequence for this method is as follows:

```
status = <object>.DecodeFrom (<inputStream>);
```

In this definition, <object> is an instance of the control class (i.e., ASN1C_<prodName>) generated for the given production.

The <inputStream> placeholder represents an input stream object type. This is an object derived from an *ASN1InputStream* class.

The function result variable `stat` returns the completion status. Error status codes are negative. Return status values are defined in the *asn1type.h* include file.

Another way to decode message using the C++ class interface is to use the >> stream operator:

```
<inputStream> >> <object>;
```

In this case, the exception *OSCStreamException* may be thrown if an error occurs.

Also, the method *Decode* without parameters is supported for backward compatibility. In this case it is necessary to create a control class object (i.e., ASN1C_<prodName>) using an input stream reference as the first parameter and a reference to a variable of the generated type as the second parameter of the constructor.

Procedure for Using the Stream-Oriented C++ Control Class Decode Method

Normally the receiving message can be one of several different message types. It is therefore necessary to determine the type of message that was received so that the appropriate decode function can be called to decode it. The *ASN1BERInputStream* class has standard methods for parsing the initial tag/length from a message to determine the type of message received. These calls are used in conjunction with a switch statement on generated tag constants for the known message set. Each switch case statement contains logic to create an object instance of a specific ASN1C generated control class and to invoke and then to invoke that object's decode method.

A program fragment that could be used to decode an employee record is as follows:

```
#include employee.h         // include file generated by ASN1C

main ()
{
   ASN1TAG tag;
   int i, len;
   const char*  filename = "message.dat";
   ASN1BOOL     trace = TRUE;

   // Decode

   ASN1BERFileInputStream in (filename);
   if (in.getStatus () != ASN_OK) {
      in.printErrorInfo ();
      return -1;
   }

   if (in.mark (32) != ASN_OK) {
```

```
         printf ("mark failed\n");
         in.printErrorInfo ();
         return -1;
      }
      if (in.decodeTagAndLen (tag, len) != ASN_OK) {
         printf ("decodeTagAndLen failed\n");
         in.printErrorInfo ();
         return -1;
      }
      if (in.reset () != ASN_OK) {
         printf ("reset failed\n");
         in.printErrorInfo ();
         return -1;
      }

      // Now switch on initial tag value to determine what
      // type of message was received..

      switch (msgtag)
      {
         case TV_PersonnelRecord:  // compiler generated
                                   // constant
         {
            ASN1T_PersonnelRecord msgData;
            ASN1C_PersonnelRecord employee (msgData);

            in >> employee;
            if (in.getStatus () != ASN_OK) {
               printf ("decode of PersonnelRecord failed\n");
               in.printErrorInfo ();
               return -1;
            }

            // or employee.DecodeFrom (in);
            break;
         }

         case TV_ ...// handle other known messages
            ...
         }
      }

      return 0;
   }
```

Note that the call to free memory and the stream's close method are not required to release dynamic memory when using the C++ interface. This is because the control class hides all of the details of managing the context and releasing dynamic memory. The memory is automatically released when both the input stream object (*ASN1BERInputStream* and derived classes) and the control class object (*ASN1C_<ProdName>*) are deleted or go out of scope. Reference counting of a context variable shared by both interfaces is used to accomplish this.

Decoding a Series of Messages Using the C++ Control Class Interface

The above example is fine as a sample for decoding a single message, but what happens in the more typical scenario of having a long-running loop that continuously decodes messages? The logic shown above would not be optimal from a performance standpoint because of the constant creation and destruction of the message processing objects. It would be much better to create all of the required objects outside of the loop and then reuse them to decode and process each message.

A code fragment showing a way to do this is as follows:

```
#include employee.h         // include file generated by ASN1C

int main ()
{
   ASN1TAGtag;
   int         i, len;
   const char*  filename = "message.dat";
   ASN1BOOL     trace = TRUE;

   // Decode

   ASN1BERFileInputStream in (filename);
   if (in.getStatus () != ASN_OK) {
      in.printErrorInfo ();
      return -1;
   }

   ASN1T_PersonnelRecord msgData;
   ASN1C_PersonnelRecord employee (msgData);

   for (;;) {
      if (in.mark (32) != ASN_OK) {
         printf ("mark failed\n");
         in.printErrorInfo ();
         return -1;
      }
      if (in.decodeTagAndLen (tag, len) != ASN_OK) {
         printf ("decodeTagAndLen failed\n");
         in.printErrorInfo ();
         return -1;
      }
      if (in.reset () != ASN_OK) {
         printf ("reset failed\n");
         in.printErrorInfo ();
         return -1;
      }

      // Now switch on initial tag value to determine what
      // type of message was received..
```

```
      switch (msgtag)
      {
         case TV_PersonnelRecord:  // compiler generated
                                   // constant
         {
            in >> employee;
            if (in.getStatus () != ASN_OK) {
               printf ("decode of PersonnelRecord failed\n");
               in.printErrorInfo ();
               return -1;
            }

            // or employee.DecodeFrom (in);

         }

         case TV_ ...// handle other known messages
            ...
      }

      // Need to reinitialize objects for next iteration

      employee.memFreeAll ();

   }  // end of loop

   return 0;
}
```

This is quite similar to the first example. Note that we have pulled the *ASN1T_Employee* and *ASN1C_Employee* object creation logic out of the switch statement and moved it above the loop. These objects can now be reused to process each received message.

The only other change was the call to *employee.memFreeAll* that was added at the bottom of the loop. Since the objects are not deleted to automatically release allocated memory, we need to do it manually. This call will free all memory held within the decoding context. This will allow the loop to start again with no outstanding memory allocations for the next pass.

Generated PER Encode Functions

PER encode/decode functions are generated when the '-per' switch is specified on the command line. For each ASN.1 production defined in the ASN.1 source file, a C PER encode function is generated. This function will convert a populated C variable of the given type into a PER encoded ASN.1 message.

If C++ code generation is specified, a control class is generated that contains an *Encode* method that wraps this function. This function is invoked through the class interface to encode an ASN.1 message into the variable referenced in the msgData component of the class.

Generated C Function Format and Calling Parameters

The format of the name of each generated PER encode function is as follows:

```
asn1PE_[<prefix>]<prodName>
```

where <prodName> is the name of the ASN.1 production for which the function is being generated and <prefix> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each encode function is as follows:

```
status = asn1PE_<name> (ASN1CTXT* pctxt, <name>[*] value);
```

In this definition, <name> denotes the prefixed production name defined above.

The pctxt argument is used to hold a context pointer to keep track of encode parameters. This is a basic "handle" variable that is used to make the function reentrant so it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her program.

The value argument contains the value to be encoded or holds a pointer to the value to be encoded. This variable is of the type generated from the ASN.1 production. The object is passed by value if it is a primitive ASN.1 data type such as BOOLEAN, INTEGER, ENUMERATED, etc.. It is passed using a pointer reference if it is a structured ASN.1 type value. Check the generated function prototype in the header file to determine how the value argument is to be passed for a given function.

The function result variable stat returns the status of the encode operation. Status code 0 (ASN_OK) indicates the function was successful. Note that this return value differs from that of BER encode functions in that the encoded length of the message component is not returned – only an OK status indicating encoding was successful. A negative value indicates encoding failed. Return status values are defined in the "asn1type.h" include file. The error text and a stack trace can be displayed using the *rtErrPrint* function.

Generated C++ Encode Method Format and Calling Parameters

Generated encode functions are invoked through the class interface by calling the base class *Encode* method. The calling sequence for this method is as follows:

```
stat = <object>.Encode ();
```

In this definition, <object> is an object of the class generated for the given production. The function result variable `stat` returns the status value from the PER encode function. This status value will be ASN_OK (0) if encoding was successful or a negative error status value if encoding fails. Return status values are defined in the "asn1type.h" include file.

The user must call the encode buffer class methods *getMsgPtr* and *getMsgLen* to obtain the starting address and length of the encoded message component.

Populating Generated Structure Variables for Encoding

See the section Populating Generated Structure Variables for Encoding for a discussion on how to populate variables for encoding. There is no difference in how it is done for BER versus how it is done for PER.

Procedure for Calling C Encode Functions

This section describes the step-by-step procedure for calling a C PER encode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

Before a PER encode function can be called, the user must first initialize an encoding context block structure. The context block is initialized by either calling the *rtNewContext* function (to allocate a dynamic context block), or by calling *rtInitContext* to initialize a static block. The user then must call *pu_setBuffer* to specify a message buffer to receive the encoded message. Specification of a dynamic message buffer is possible by setting the buffer address argument to null and the buffer size argument to zero. The *pu_setBuffer* function also allows for the specification of aligned or unaligned encoding.

An encode function can then be called to encode the message. If the return status indicates success (ASN_OK), then the message will have been encoded in the given buffer. PER encoding starts from the beginning of the buffer and proceeds from low memory to high memory until the message is complete. This differs from BER where encoding was done from back-to-front. Therefore, the buffer start address is where the encoded PER message begins. The length of the encoded message can be obtained by calling the *pe_GetMsgLen* run-time function. If dynamic encoding was specified (i.e., a buffer start address and length were not given), the run-time routine *pe_GetMsgPtr* can be used to obtain the start address of the message. This routine will also return the length of the encoded message.

A program fragment that could be used to encode an employee record is as follows:

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
   ASN1OCTET msgbuf[1024];
   int       msglen, stat;
   ASN1CTXT  ctxt;
   ASN1BOOL  aligned = TRUE;
   Employee  employee; /* typedef generated by ASN1C */

   /* Populate employee C structure */
```

```
   employee.name.givenName = "SMITH";
   ...

   /* Allocate and initialize a new context pointer */

   stat = rtInitContext (&ctxt);

   if (stat != ASN_OK) {
      printf ("rtInitContext failed (check license)\n");
      rtErrPrint (&ctxt);
      return stat;
   }

   pu_setBuffer (&ctxt, msgbuf, msglen, aligned);

   if ((stat = asn1PE_Employee (&ctxt, &employee)) == ASN_OK) {
      msglen = pe_GetMsgLen (&ctxt);
      ...
   }
   else
      error processing...
}
```

In general, static buffers should be used for encoding messages where possible as they offer a substantial performance benefit over dynamic buffer allocation. The problem with static buffers, however, is that you are required to estimate in advance the approximate size of the messages you will be encoding. There is no built-in formula to do this, the size of an ASN.1 message can vary widely based on data types and other factors.

If performance is not a significant issue, then dynamic buffer allocation is a good alternative. Setting the buffer pointer argument to NULL in the call to *pu_setBuffer* specifies dynamic allocation. This tells the encoding functions to allocate a buffer dynamically. The address of the start of the message is obtained after encoding by calling the run-time function *pe_GetMsgPtr*.

The following code fragment illustrates PER encoding using a dynamic buffer:

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
   ASN1OCTET *msgptr;
   int       msglen, stat;
   ASN1CTXT  ctxt;
   ASN1BOOL  aligned = TRUE;
   Employee  employee;/* typedef generated by ASN1C */

   employee.name.givenName = "SMITH";
   ...

   stat = rtInitContext (&ctxt);

   if (stat != ASN_OK) {
      printf ("rtInitContext failed (check license)\n");
      rtErrPrint (&ctxt);
      return stat;
   }

   pu_setBuffer (&ctxt, 0, 0, aligned);

   if ((stat = asn1PE_Employee (&ctxt, &employee)) == ASN_OK) {
      msgptr = pe_GetMsgPtr (&ctxt, &msglen);
      ...

   }
   else
      error processing...
}
```

Procedure for Using the C++ Control Class Encode Method

The procedure to encode a message using the C++ class interface is as follows:

1. Instantiate an ASN.1 PER encode buffer object (*ASN1PEREncodeBuffer*) to describe the buffer into which the message will be encoded. Two overloaded constructors are available. The first form takes as arguments a static encode buffer and size and a Boolean value indicating whether aligned encoding is to be done. The second form only takes the Boolean aligned argument. This form is used to specify dynamic encoding.

2. Instantiate an ASN1T_<ProdName> object and populate it with data to be encoded.

3. Instantiate an ASN1C_<ProdName> object to associate the message buffer with the data to be encoded.

4. Invoke the ASN1C_<ProdName> object Encode method.

5. Check the return status. The return value is a status value indicating whether encoding was successful or not. Zero (ASN_OK) indicates success. If encoding failed, the status value will be a negative number. The encode buffer method *printErrorInfo* can be invoked to get a textual explanation and stack trace of where the error occurred.

6. If encoding was successful, get the start-of-message pointer and message length. The start-of-message pointer is obtained by calling the *getMsgPtr* method of the encode buffer object. If static encoding was specified (i.e., a message buffer address and size were specified to the PER Encode Buffer class constructor), the start-of-message pointer is the buffer start address. The message length is obtained by calling the *getMsgLen* method of the encode buffer object.

A program fragment that could be used to encode an employee record is as follows:

```
#include employee.h         // include file generated by ASN1C

main ()
{
   const ASN1OCTET* msgptr;
   ASN1OCTET msgbuf[1024];
   int       msglen, stat;
   ASN1BOOL  aligned = TRUE;

   // step 1: instantiate an instance of the PER encode
   // buffer class.  This example specifies a static
   // message buffer..

   ASN1PEREncodeBuffer encodeBuffer (msgbuf,
                                     sizeof(msgbuf),
                                     aligned);

   // step 2: populate msgData with data to be encoded

   ASN1T_PersonnelRecord msgData;
   msgData.name.givenName = "SMITH";
   ...

   // step 3: instantiate an instance of the ASN1C_<ProdName>
   // class to associate the encode buffer and message data..

   ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

   // steps 4 and 5: encode and check return status

   if ((stat = employee.Encode ()) == ASN_OK)
   {
      printf ("Encoding was successful\n");
      printf ("Hex dump of encoded record:\n");
      encodeBuffer.hexDump ();
      printf ("Binary dump:\n");
      encodeBuffer.binDump ("employee");

      // step 6: get start-of-message pointer and message length.
      // start-of-message pointer is start of msgbuf
      // call getMsgLen to get message length..

      msgptr = encodeBuffer.getMsgPtr ();  // will return &msgbuf
      len = encodeBuffer.getMsgLen ();
   }
   else
   {
      printf ("Encoding failed\n");
      encodeBuffer.printErrorInfo ();
```

```
      exit (0);
   }

   // msgptr and len now describe fully encoded message

   ...
```

In general, static buffers should be used for encoding messages where possible as they offer a substantial performance benefit over dynamic buffer allocation. The problem with static buffers, however, is that you are required to estimate in advance the approximate size of the messages you will be encoding. There is no built-in formula to do this, the size of an ASN.1 message can vary widely based on data types and other factors.

If performance is not a significant issue, then dynamic buffer allocation is a good alternative. Using the form of the *ASN1PEREncodeBuffer* constructor that does not include buffer address and size arguments specifies dynamic buffer allocation. This constructor only requires a Boolean value to specify whether aligned or unaligned encoding should be performed (aligned is true).

The following code fragment illustrates PER encoding using a dynamic buffer:

```
#include employee.h         // include file generated by ASN1C

main ()
{
   ASN1OCTET *msgptr;
   int       msglen, stat;
   ASN1BOOL  aligned = TRUE;

   // Create an instance of the compiler generated class.
   // This example does dynamic encoding (no message buffer
   // is specified)..

   ASN1PEREncodeBuffer encodeBuffer (aligned);
   ASN1T_PersonnelRecord msgData;
   ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

   // Populate msgData within the class variable

   msgData.name.givenName = "SMITH";
   ...

   // Encode

   if ((stat = employee.Encode ()) == ASN_OK)
   {
      printf ("Encoding was successful\n");
      printf ("Hex dump of encoded record:\n");
      encodeBuffer.hexDump ();
      printf ("Binary dump:\n");
      encodeBuffer.binDump ("employee");
```

```
      // Get start-of-message pointer and length

      msgptr = encodeBuffer.getMsgPtr ();
      len = encodeBuffer.getMsgLen ();
   }
   else
   {
      printf ("Encoding failed\n");
      encodeBuffer.printErrorInfo ();
      exit (0);
   }

   return 0;
}
```

Encoding a Series of PER Messages using the C++ Interface

When encoding a series of PER messages using the C++ interface, performance can be improved by reusing the message processing objects to encode each message rather than creating and destroying the objects each time. A detailed example of how to do this was given in the section on BER message encoding. The PER case would be similar with the PER function calls substituted for the BER calls. As was the case for BER, the encode message buffer object *init* method can be used to reinitialize the encode buffer between invocations of the encode functions.

Generated PER Decode Functions

PER encode/decode functions are generated when the '-per' switch is specified on the command line. For each ASN.1 production defined in the ASN.1 source file, a C PER decode function is generated. This function will parse the data contents from a PER-encoded ASN.1 message and populate a variable of the corresponding type with the data.

If C++ code generation is specified, a control class is generated that contains a Decode method that wraps this function. This function is invoked through the class interface to encode an ASN.1 message into the variable referenced in the msgData component of the class.

Generated C Function Format and Calling Parameters

The format of the name of each generated PER decode function is as follows:

```
asn1PD_[<prefix>]<prodName>
```

where <prodName> is the name of the ASN.1 production for which the function is being generated and <prefix> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each decode function is as follows:

```
status = asn1PD_<name> (ASN1CTXT* pctxt, <name>* pvalue);
```

In this definition, <name> denotes the prefixed production name defined above.

The `pctxt` argument is used to hold a context pointer to keep track of decode parameters. This is a basic "handle" variable that is used to make the function reentrant so it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her program.

The `pvalue` argument is a pointer to a variable to hold the decoded result. This variable is of the type generated from the ASN.1 production. The decode function will automatically allocate dynamic memory for variable length fields within the structure. This memory is tracked within the context structure and is released when the context structure is freed.

The function result variable `stat` returns the status of the decode operation. Status code 0 (ASN_OK) indicates the function was successful. A negative value indicates decoding failed. Return status values are defined in the "asn1type.h" include file. The reason text and a stack trace can be displayed using the rtErrPrint function described later in this document.

Generated C++ Decode Method Format and Calling Parameters

Generated decode functions are invoked through the class interface by calling the base class *Decode* method. The calling sequence for this method is as follows:

```
status = <object>.Decode ();
```

In this definition, <object> is an object of the class generated for the given production.

An *ASN1PERDecodeBuffer* object must be passed to the <object> constructor prior to decoding. This is where the message start address and length are specified. A Boolean argument is also passed indicating whether the message to be decoded was encoded using aligned or unaligned PER

The function result variable `status` returns the status of the decode operation. The return status will be zero (ASN_OK) if decoding is successful or a negative value if an error occurs. Return status values are defined in Appendix A of the C/C++ Common Functions Reference Manual of this document and online in the *asn1type.h* include file.

Procedure for Calling C Decode Functions

This section describes the step-by-step procedure for calling a C PER decode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

Unlike BER, the user must know the ASN.1 type of a PER message before it can be decoded. This is because the type cannot be determined at run-time. There are no embedded tag values to reference to determine the type of message received.

There are three steps to calling a compiler-generated decode function:

1. Prepare a context variable for decoding
2. Call the appropriate compiler-generated decode function to decode the message
3. Free the context after use of the decoded data is complete to free allocated memory structures

Before a PER decode function can be called, the user must first initialize a context block structure. The context block is initialized by either calling the *rtNewContext* function (to allocate a dynamic context block), or by calling *rtInitContext* to initialize a static block. The *pu_setBuffer* function must them be called to specify a message buffer that contains the PER-encoded message to be decoded. This function also allows for the specification of aligned or unaligned decoding.

A decode function can then be called to decode the message. If the return status indicates success (ASN_OK), then the message will have been decoded into the given ASN.1 type variable. The decode function may automatically allocate dynamic memory to hold variable length variables during the course of decoding. This memory will be tracked in the context structure, so the programmer does not need to worry about freeing it. It will be released when the context is freed.

The final step of the procedure is to free the context block. This must be done regardless of whether the block is static (declared on the stack and initialized using *rtInitContext*), or dynamic (created using *rtNewContext*). The function to free the context is *rtFreeContext*.

A program fragment that could be used to decode an employee record is as follows:

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
    ASN1OCTET msgbuf[1024];
    ASN1TAG   msgtag;
    int       msglen, stat;
    ASN1CTXT  ctxt;
    ASN1BOOL  aligned = TRUE;
    PersonnelRecord employee;

    .. logic to read message into msgbuf ..

    /* This example uses a static context block */

    /* step 1: prepare the context block */

    stat = rtInitContext (&ctxt);

    if (stat != ASN_OK) {
       printf ("rtInitContext failed (check license)\n");
       rtErrPrint (&ctxt);
       return stat;
    }

    pu_setBuffer (&ctxt, msgbuf, msglen, aligned);

    /* step 2: decode the record */

    stat = asn1PD_PersonnelRecord (&ctxt, &employee);

    if (stat == ASN_OK)
    {
        process received data..
```

```
    }
    else {
        /* error processing... */
        rtErrPrint (&ctxt);
    }

    /* step 3: free the context */

    rtFreeContext (&ctxt);
}
```

Procedure for Using the C++ Control Class Encode Method

The following are the steps are involved in decoding a PER message using the generated C++ class:

1. Instantiate an ASN.1 PER decode buffer object (*ASN1PERDecodeBuffer*) to describe the message to be decoded. The constructor takes as arguments a pointer to the message to be decoded, the length of the message, and a flag indicating whether aligned encoding was used or not.

2. Instantiate an ASN1T_<ProdName> object to hold the decoded message data.

3. Instantiate an ASN1C_<ProdName> object to decode the message. This class associates the message buffer object with the object that is to receive the decoded data. The results of the decode operation will be placed in the variable declared in step 2.

4. Invoke the ASN1C_<ProdName> object Decode method.

5. Check the return status. The return value is a status value indicating whether decoding was successful or not. Zero (ASN_OK) indicates success. If decoding failed, the status value will be a negative number. The decode buffer method 'PrintErrorInfo' can be invoked to get a textual explanation and stack trace of where the error occurred.

6. Release dynamic memory that was allocated by the decoder. All memory associated with the decode context is released when both the ASN1PERDecodeBuffer and ASN1C_<ProdName> objects go out of scope.

A program fragment that could be used to decode an employee record is as follows:

```
#include employee.h         // include file generated by ASN1CPP

main ()
{
    ASN1OCTET msgbuf[1024];
    int       msglen, stat;
    ASN1BOOL  aligned = TRUE;

    .. logic to read message into msgbuf ..

    // step 1: instantiate a PER decode buffer object

    ASN1PERDecodeBuffer decodeBuffer (msgbuf, msglen, aligned);

    // step 2: instantiate an ASN1T_<ProdName> object
```

```
    ASN1T_PersonnelRecord msgData;

    // step 3: instantiate an ASN1C_<ProdName> object

    ASN1C_PersonnelRecord employee (decodeBuffer, msgData);

    // step 4: decode the record

    stat = employee.Decode ();

    // step 5: check the return status

    if (stat == ASN_OK)
    {
        process received data..
    }
    else {
        // error processing..
        decodeBuffer.PrintErrorInfo ();
    }

    // step 6: free dynamic memory (will be done automatically
    // when both the decodeBuffer and employee objects go out
    // of scope)..

}
```

Decoding a Series of Messages Using the C++ Control Class Interface

The above example is fine as a sample for decoding a single message, but what happens in the more typical scenario of having a long-running loop that continuously decodes messages? The logic shown above would not be optimal from a performance standpoint because of the constant creation and destruction of the message processing objects. It would be much better to create all of the required objects outside of the loop and then reuse them to decode and process each message.

A code fragment showing a way to do this is as follows:

```
#include employee.h         // include file generated by ASN1C
main ()
{
    ASN1OCTET msgbuf[1024];
    int       msglen, stat;
    ASN1BOOL  aligned = TRUE;

    // step 1: instantiate a PER decode buffer object

    ASN1PERDecodeBuffer decodeBuffer (msgbuf, msglen, aligned);

    // step 2: instantiate an ASN1T_<ProdName> object

    ASN1T_PersonnelRecord msgData;

    // step 3: instantiate an ASN1C_<ProdName> object

    ASN1C_PersonnelRecord employee (decodeBuffer, msgData);

    // loop to continuously decode records

    for (;;) {
        .. logic to read message into msgbuf ..

        stat = employee.Decode ();
        // step 5: check the return status

        if (stat == ASN_OK)
        {
            process received data..
        }
        else {
            // error processing..
            decodeBuffer.PrintErrorInfo ();
        }

        // step 6: free dynamic memory

        employee.memFreeAll ();
    }
}
```

The only difference between this and the previous example is the addition of the decoding loop and the modification of step 6 in the procedure. The decoding loop is an infinite loop to continuously read and decode messages from some interface such as a network socket. The decode calls are the same, but before in step 6, we were counting on the message buffer and control objects to go out of scope to cause the memory to be released. Since the objects are now being reused, this will not happen. So the call to the *memFreeAll* method that is defined in the ASN1C_Type base class will force all memory held at that point to be released.

Performance Considerations: Dynamic Memory Management

Please refer Performance Considerations: Dynamic Memory Management in the *BER Decode Functions* section for a discussion of memory management performance issues. All of the issues that apply to BER and DER also apply to PER as well.

Generated XER Encode Functions

XER C encode functions are generated when the '-xer' switch is specified on the command line. For each ASN.1 production defined in the ASN.1 source file, a C XER encode function is generated. This function will convert a populated C variable of the given type into an XER encoded ASN.1 message (i.e. an XML document).

If C++ code generation is specified, a control class is generated that contains an Encode method that wraps this function. This function is invoked through the class interface to encode an ASN.1 message into the variable referenced in the msgData component of the class.

Generated C Function Format and Calling Parameters

The format of the name of each generated XER encode function is as follows:

```
asn1XE_[<prefix>]<prodName>
```

where <prodName> is the name of the ASN.1 production for which the function is being generated and <prefix> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each encode function is as follows:

```
status = asn1XE_<name> (ASN1CTXT* pctxt, <name>[*] value,
                        ASN1ConstCharPtr elemName,
                        ASN1ConstCharPtr attributes);
```

In this definition, <name> denotes the prefixed production name defined above.

The `pctxt` argument is used to hold a context pointer to keep track of encode parameters. This is a basic "handle" variable that is used to make the function reentrant so it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her or her program.

The `value` argument contains the value to be encoded or holds a pointer to the value to be encoded. This variable is of the type generated from the ASN.1 production. The object is passed by value if it is a primitive ASN.1 data type such as BOOLEAN, INTEGER, ENUMERATED, etc.. It is passed using a pointer reference if it is a structured ASN.1 type value (in this case, the name will be pvalue instead of value). Check the generated function prototype in the header file to determine how this argument is to be passed for a given function.

The *elemName* and *attributes* arguments are used to pass the XML element name and attributes respectively. The *elemName* argument is the name that will be included in the <name> </name> brackets used to delimit an XML item. There are three distinct ways this argument can be specified:

1. If it contains standard text, this text will be used as the element name.

2. If it is null, a default element name will be applied. Default names for all of the built-in ASN.1 types are defined in the 2002 X.680 standard. For example, <BOOLEAN> is the default element name for the BOOLEAN built-in type.

3. If the name is empty (i.e. equal to "", a zero-length string – not to be confused with null), then no element name is applied to the encoded data.

The function result variable `stat` returns the status of the encode operation. Status code 0 (ASN_OK) indicates the function was successful. A negative value indicates encoding failed. Return status values are defined in the "asn1type.h" include file. The error text and a stack trace can be displayed using the *rtErrPrint* function.

Generated C++ Encode Method Format and Calling Parameters

Generated encode functions are invoked through the class interface by calling the base class 'Encode' method. The calling sequence for this method is as follows:

```
stat = <object>.Encode ();
```

In this definition, <object> is an object of the class generated for the given production. The function result variable `stat` returns the status value from the XER encode function. This status value will be ASN_OK (0) if encoding was successful or a negative error status value if encoding fails. Return status values are defined in the "asn1type.h" include file.

The user must call the encode buffer class methods *getMsgPtr* and *getMsgLen* to obtain the starting address and length of the encoded message component.

Populating Generated Structure Variables for Encoding

See the section Populating Generated Structure Variables for Encoding for a discussion on how to populate variables for encoding. There is no difference in how it is done for any of the different encoding rules.

Procedure for Calling C Encode Functions

This section describes the step-by-step procedure for calling C XER encode functions. This procedure is similar to that for the other encoding methods except that some of the functions used are specific to XER.

Before an XER encode function can be called, the user must first initialize an encoding context block structure. The context block is initialized by calling the *rtInitContext* function. This routine allows a message buffer to be specified to receive the encoded message. Specification of a message buffer is optional. If not specified, the encoder will allocate memory automatically for the encoded message.

An encode function can then be called to encode the message. If the return status indicates success (ASN_OK), then the message will have been encoded in the given buffer. XER encoding starts from the beginning of the buffer and proceeds from low memory to high memory until the message is complete. This differs from BER where encoding was done from back-to-front. Therefore, the buffer start address is where the encoded XER message begins. The length of the encoded message can be obtained by calling the *xerGetMsgLen* run-time function. If dynamic encoding was specified (i.e., a buffer start address and length were not given), the run-time routine *xerGetMsgPtr* can be used to obtain the start address of the message. This routine will also return the length of the encoded message.

A program fragment that could be used to encode an employee record is as follows:

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
    ASN1OCTET msgbuf[1024];
    int       msglen, stat;
    ASN1CTXT  ctxt;
    ASN1BOOL  cxer = FALSE;  /* canonical XER flag */
    ASN1BOOL  aligned = TRUE;
    Employee  employee;/* typedef generated by ASN1C */

    /* Initialize context and set encode buffer pointer */
    if (rtInitContext (&ctxt) != ASN_OK) return -1;

    xerSetEncBufPtr (&ctxt, msgbuf, sizeof(msgbuf), cxer);

    /* Populate variable with data to be encoded */
    employee.name.givenName = "John";
    ...

    /* Encode data */
    stat = asn1XE_Employee (&ctxt, &employee, 0, 0);

    if (stat) == ASN_OK) {
      msglen = xerGetMsgLen (&ctxt);
      ...
    }
    else
      error processing...
}

rtFreeContext (&ctxt);  /* release the context pointer */
```

After encoding is complete, msgbuf contains the XML textual representation of the data. By default, a UTF-8 encoding is used. For the ASCII character set, this results in a buffer containing normal textual data. Therefore, the contents of the buffer are represented as a normal text string and can be displayed using the C *printf* run-time function or any other function capable of displaying text.

Procedure for Using the C++ Control Class Encode Method

The procedure to encode a message using the C++ class interface is as follows:

1. Instantiate an ASN.1 XER encode buffer object (ASN1XEREncodeBuffer) to describe the buffer into which the message will be encoded. Constructors are available that allow a static message buffer to be specified and/or canonical encoding to be turned on (canonical encoding removes all encoding options from the final message to produce a single encoded representation of the data). The default constructor specifies use of a dynamic encode buffer and canonical encoding set to off.

2. Instantiate an ASN1T_<type> object and populate it with data to be encoded.

3. Instantiate an ASN1C_<type> object to associate the message buffer with the data to be encoded.

4. Invoke the ASN1C_<type> object Encode method.

5. Check the return status. The return value is a status value indicating whether encoding was successful or not. Zero (ASN_OK) indicates success. If encoding failed, the status value will be a negative number. The encode buffer method *printErrorInfo* can be invoked to get a textual explanation and stack trace of where the error occurred.

6. If encoding was successful, get the start-of-message pointer and message length. The start-of-message pointer is obtained by calling the *getMsgPtr* method of the encode buffer object. If static encoding was specified (i.e., a message buffer address and size were specified to the PER Encode Buffer class constructor), the start-of-message pointer is the buffer start address. The message length is obtained by calling the *getMsgLen* method of the encode buffer object.

A program fragment that could be used to encode an employee record is as follows:

```
#include employee.h         // include file generated by ASN1C

main ()
{
    const ASN1OCTET* msgptr;
    ASN1OCTET msgbuf[1024];
    int       msglen, stat;
    ASN1BOOL  canonical = FALSE;

    // step 1: instantiate an instance of the XER encode
    // buffer class.  This example specifies a static
    // message buffer..

    ASN1XEREncodeBuffer encodeBuffer (msgbuf,
                                      sizeof(msgbuf),
                                      canonical);

    // step 2: populate msgData with data to be encoded

    ASN1T_PersonnelRecord msgData;
    msgData.name.givenName = "SMITH";
    ...
```

```
// step 3: instantiate an instance of the ASN1C_<ProdName>
// class to associate the encode buffer and message data..

ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

// steps 4 and 5: encode and check return status

if ((stat = employee.Encode ()) == ASN_OK)
{
   printf ("encoded XML message:\n");
   printf ((ASN1ConstCharPtr)msgbuf);
   printf ("\n");

   // step 6: get start-of-message pointer and message length.
   // start-of-message pointer is start of msgbuf
   // call getMsgLen to get message length..

   msgptr = encodeBuffer.getMsgPtr ();  // will return &msgbuf
   len = encodeBuffer.getMsgLen ();
}
else
{
   printf ("Encoding failed\n");
   encodeBuffer.printErrorInfo ();
   exit (0);
}

// msgptr and len now describe fully encoded message

...
```

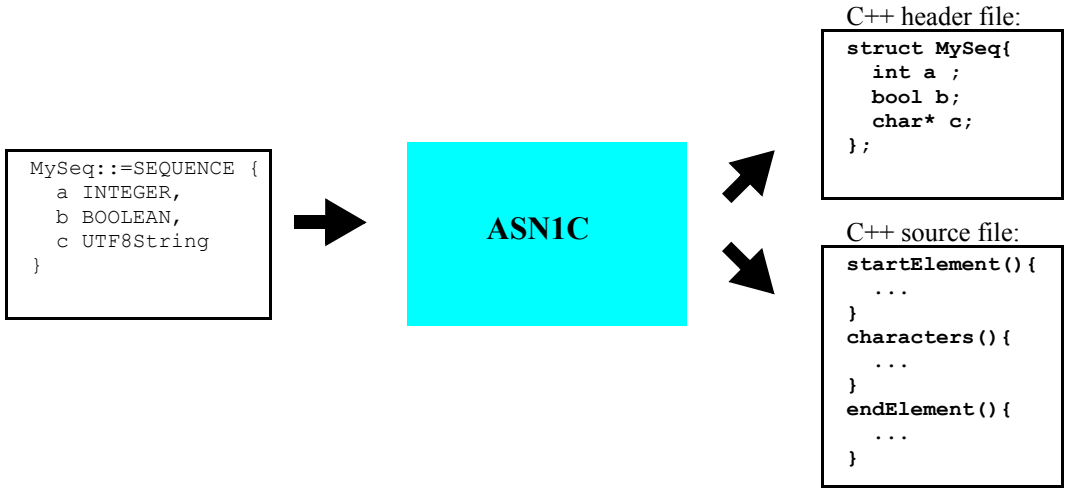
Generated XER Decode Functions

The code generated to decode XML messages is different than that of the other encoding rules. This is because off-the-shelf XML parser software is used to parse the XML documents to be decoded. This software contains a common interface known as the *Simple API for XML (or SAX)* that is a de-facto standard that is supported by most parsers. ASN1C generates an implementation of the content handler interface defined by this standard. This implementation receives the parsed XML data and uses it to populate the structures generated by the compiler.

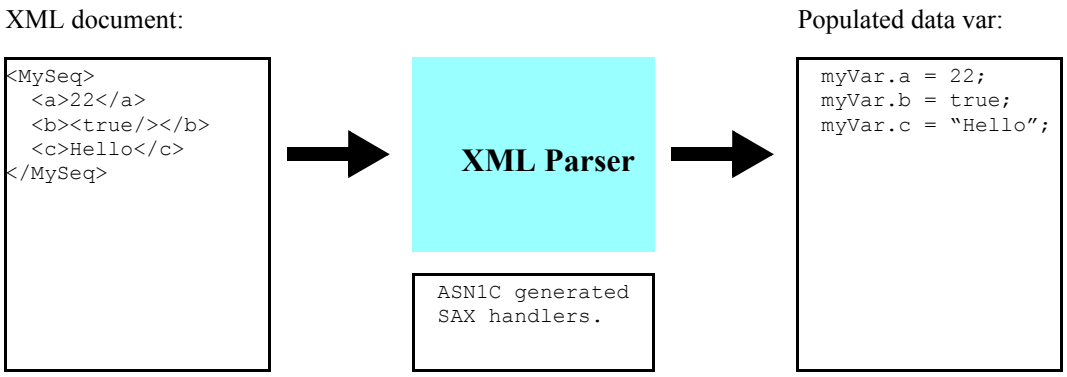
The default XML parser used is the EXPAT parser (http://www.expat.org). This is a lightweight, open-source parser that was implemented in C. Objective Systems added the C++ SAX interface by adapting the headers of the Apache XERCES C++ XML Parser (http://xml.apache.org) to work with the underlying C code. These headers were used to build a common C++ SAX interface across different vendor's SAX interfaces (unlike Java, these interfaces are not all the same). The ASN1C XER SAX C and C++ libraries come with the EXPAT parser as the default parser and also include plug-in interfaces that allow the code to work with the Microsoft XML parser (MSXML), The GNOME libxml2 parser, and the XERCES XML parser. Interfacing to other parsers only requires building an abstraction layer to map the common interface to the vendor's interface.

A diagram showing the components used in the XML decode process is as follows:

Step 1: Generate code:



Step 2: Build Application:



ASN1C generates code to implement the following methods defined in the SAX content handler interface:

```
startElement

characters

endElement
```

The interface defines other methods that can be implemented as well, but these are sufficient to decode XER encoded data.

Procedure for Using the C Interface

The ASN1C compiler generates XER decode functions for C for constructed types in a specification. These can be invoked in the same manner as other decode functions. In this case, they install the generated SAX content handler functions and invoke the XML parser's *parse* function to parse a document. The procedure to call these generated functions is described below.

Generated C Function Format and Calling Parameters

The format of the name of each generated C XER decode function is as follows:

```
asn1XD_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each decode function is as follows:

```
status = asn1XD_<name> (ASN1CTXT* pctxt, <name>* pvalue);
```

In this definition, <name> denotes the prefixed production name defined above.

The `pctxt` argument is used to hold a context pointer to keep track of decode parameters. This is a basic "handle" variable that is used to make the function reentrant so that it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her program. The variable must be initialized using the *rtInitContext* run-time function before use.

C XER decoding is stream-oriented. To perform streaming operations, the context pointer *pctxt* must also be initialized as a stream by using the *rtStreamInit* run-time library function (see the *C/C++ Common Run-Time Library Reference Manual* for a description of the run-time stream C functions).

The `pvalue` argument is a pointer to a variable to hold the decoded result. This variable is of the type generated from the ASN.1 production. The decode function will automatically allocate dynamic memory for variable length fields within the structure. This memory is tracked within the context structure and is released when the context structure is freed.

The function result variable `stat` returns the status of the decode operation. Status code 0 (ASN_OK) indicates the function was successful. A negative value indicates decoding failed. Return status values are defined in the "asn1type.h" include file. The reason text and a stack trace can be displayed using the *rtErrPrint* function.

Procedure for Calling C Decode Functions

This section describes the step-by-step procedure for calling a C XER decode function. This method must be used if C code generation was done. This method cannot be used as an alternative to using the control class interface if C++ code generation was done. Use the C++ procedure instead.

There are four steps to calling a compiler-generated decode function:

1. Prepare a context variable for decoding;
2. Open a stream;
3. Call the appropriate compiler-generated decode function to decode the message;
4. Free the context after use of the decoded data is complete to free allocated memory structures

Before a C XER decode function can be called; the user must initialize a context variable. This is a variable of type ASN1CTXT. This variable holds all of the working data used during the decoding of a message. The context variable is declared as a normal automatic variable within the top-level calling function. **It must be initialized before use**. This can be accomplished by using the *rtInitContext* function. Also, the context must be initialized for streaming operations by calling the *rtStreamInit* function:

```
ASN1CTXT ctxt; // context variable

if (rtInitContext (&ctxt) != ASN_OK) {
   /* initialization failed, could be a license problem */
   printf ("context initialization failed (check license)\n");
   return -1;
}

rtStreamInit (&ctxt));// Initialize stream
```

The next step is to create a stream object within the context. This object is an abstraction of the output device to which the data is to be encoded and is initialized by calling one of the following functions:

- *rtStreamFileOpen*
- *rtStreamFileAttach*
- *rtStreamSocketAttach*
- *rtStreamMemoryCreate*
- *rtStreamMemoryAttach*

The *flags* parameter of these functions should be set to the OSRTSTRMF_INPUT constant value to indicate an input stream is being created (see the *C/C++ Common Run-Time Library Reference Manual* for a full description of these functions).

A decode function can then be called to decode the message. If the return status indicates success (ASN_OK), then the message will have been decoded into the given ASN.1 type variable. The decode function may automatically allocate dynamic memory to hold variable length items during the course of decoding. This memory will be tracked in the

context structure, so the programmer does not need to worry about freeing it. It will be released when the context is freed.

The final step of the procedure is to close the stream and free the context block. The function to free the context is *rtFreeContext*.

A program fragment that could be used to decode an employee record is as follows:

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
    int       stat;
    ASN1CTXT  ctxt;
    PersonnelRecord employee;
    ASN1ConstCharPtr filename = "message.xml";

    /* Step 1: Init context structure */

    if (rtInitContext (&ctxt) != ASN_OK) return -1;

    rtStreamInit (&ctxt);

    /* Step 2: Open a stream */

    stat = rtStreamFileOpen (&ctxt, filename, OSRTSTRMF_INPUT);
    if (stat != ASN_OK) {
       rtErrPrint (&ctxt.errInfo);
       return -1;
    }

    /* Step 3: decode the record */

    stat = asn1XD_PersonnelRecord (&ctxt, &employee);
    if (stat == ASN_OK) {
       if (trace) {
          printf ("Decode of PersonnelRecord was successful\n");
          printf ("Decoded record:\n");
          asn1Print_PersonnelRecord ("Employee", &employee);
       }
    }
    else {
       printf ("decode of PersonnelRecord failed\n");
       rtErrPrint (&ctxt.errInfo);
       rtStreamClose (&ctxt);
       return -1;
    }

    /* Step 4: Close the stream and free the context. */

    rtStreamClose (&ctxt);
```

```
        rtFreeContext (&ctxt);

        return 0;
    }
```

Procedure for Using the C++ Interface

SAX handler methods are added to the C++ control class generated for each ASN.1 production.

The procedure to invoke the generated decode method is similar to that for the other encoding rules. It is as follows:

1. Instantiate an ASN.1 XER decode buffer object (*ASN1XERDecodeBuffer*) to describe the message to be decoded. Constructors exist that allow an XML file or memory buffer to be specified as an input source.

2. Instantiate an ASN1T_*TypeName* object to hold the decoded message data.

3. Instantiate an ASN1C_*TypeName* object to decode the message. This class associates the message buffer object with the object that is to receive the decoded data. The results of the decode operation will be placed in the variable declared in step 2.

4. Invoke the ASN1C_*TypeName* object *Decode* method. This method initiates and invokes the XML parser's *parse* method to parse the document. This, in turn, invokes the generated SAX handler methods.

5. Error handling is accomplished using a try-catch block to catch SAX exceptions. In general, ASN1C does not use exceptions because they are not portable in all situations and are inefficient in terms of performance. However, exceptions are part of the standard SAX interface, so they are used in this case.

6. Release dynamic memory that was allocated by the decoder. All memory associated with the decode context is released when both the ASN1XERDecodeBuffer and ASN1C_*TypeName* objects go out of scope.

A program fragment that could be used to decode an employee record is as follows:

```
int main (int argc, char* argv[])
{
   const char*  filename = "employee.xml";
   int          stat;

   // note that a try/catch block must be used to catch SAX
   // exceptions

   try {

      // steps 1, 2, and 3: instantiate an instance of the XER
      // decoding classes.  This example specifies an XML file
      // as the message input source..

      ASN1T_PersonnelRecord employee;
      ASN1XERDecodeBuffer decodeBuffer (filename);
      ASN1C_PersonnelRecord employeeC (decodeBuffer, employee);

      // step 4: invoke the decode method
```

```
      stat = employeeC.Decode ();

      if (ASN_OK == stat) {
         employeeC.Print ("employee");
      }
      else
         printf ("stat = %d\n", stat);
   }

   // step 5: catch SAX exceptions (note: the StrX constructor
   // transcodes Unicode characters to ASCII.  All internal
   // XML parser data is formatted as Unicode.)

   catch (const SAXException& toCatch) {
      cerr << "\nAn error occured\n  Error: "
           << StrX (toCatch.getMessage())
           << "\n" << endl;

      stat = ASN_E_XMLPARSE;
   }

   // step 6: dynamic memory is released when employeeC and
   // decode buffer objects go out of scope.

   return (stat);

}
```

Procedure for Interfacing with Other C and C++ XML Parser Libraries

As mentioned previously, the Expat XML Parser library is the default XML parser library implementation used for decoding XER messages. It is also possible to use the C++ SAX handlers generated by ASN1C with other XML parser library implementations. The XER Run-Time Library (ASN1XER) provides a common interface to other parsers via a common adapter interface layer. There is a special XML interface object file for each of the following supported XML parsers:

- RTXMLLIBXML2IF - interface to LIBXML2;
- RTXMLMSXMLIF - interface to MSXML;
- RTXMLXERCESIF - interface to XERCES;
- RTXMLEXPATIF - interface to EXPAT.

The XER Run-Time Library is completely independent from the XML readers because the adapter layer within these libraries defines a common SAX API.

If an application is linked statically then the static variant of one of these interface objects (their names have suffix "_A") should be linked cooperatively with the XML parser, ASN1XER and ASN1RT libraries.

If the application is linked dynamically (using dynamically-linked libraries (DLL) in Windows or shared objects (SO or SL in UNIX/Linux) then it is necessary to link the application with the dynamic variant of the interfaces (without suffix "_A"), dynamic version of the XML parser, ASN1XER and ASN1RT dynamic libraries.

Generated XML Encode and Decode Functions

XML C encode and decode functions are generated when the '-xml' switch is specified on the command line. These are very similar to the XER encode functions described earlier. Like XER, this function allows data in a populated variable to be formatted into an XML document. Unlike the XER variant, this function will produce XML that adheres more closely to the Worldwide Web Consortium (W3C) XML conventions. In particular, the following differences exist:

- Lists of numbers, enumerated tokens, and named bits are expressed in space-separated list form instead of as individually wrapped elements or value lists.

  For example, the ASN.1 specification "A ::= SEQUENCE OF INTEGER" with value "{ 1 2 3 }" would produce the following encoding in XER:

  <A><INTEGER>1</INTEGER><INTEGER>2</INTEGER><INTEGER>3</INTEGER></A>

  in XML, it would be the following:

  <A>1 2 3</A>

- The values of the BOOLEAN data type are expressed as the lower case words "true" or "false" with no delimiters. In XER, the values are <TRUE/> and <FALSE/>.

- Enumerated token values are expressed as the identifiers themselves instead of as empty XML elements (i.e. elements wrapped in '< />'). For example, a value of the ASN.1 type "Colors ::= ENUMERATED { red, blue, green }" equal to "red" would simply be "<color>red</color>" instead of "<color><red/></color>".

- The special REAL values <PLUS-INFINITY/> and <MINUS-INFINITY/> are represented as INF and -INF respectively.

- GeneralizedTime and UTCTime values are transformed into the XSD representation for dateTime (YYYY-MM-DDTHH:MM:SS[.SSSS][(Z|(+|-)HH:MM)]) when encoded to XML. When an XML document is decoded, the time format is transformed into the ASN.1 format.

- An unnamed element in a SEQUENCE OF construct will be wrapped with the fixed keyword "element". In XER, the non-parameterized type name is used.

  For example, in the employee sample program, the following element exists in the PersonnelRecord construct:

  ```
  children[3] IMPLICIT SEQUENCE OF ChildInformation
  ```

  In XER, this is encoded as follows:

  ```
  <children><ChildInformation>...</ChildInformation>...
  ```

  In XML, it is as follows:

  ```
  <children><element>...</element>...
  ```

  It is done this way in order to match the XML schema generated for the ASN.1 specification.

The calling conventions for generated XML encode and decode functions are identical to those of XER that were presented in the previous section. The only difference is the prefix of the generated functions. For XER, encode functions contain the "asn1XE_" prefix and decode functions have "asn1XD_". For XML, the prefixes are "asn1XMLE_" and "asn1XMLD_" respectively.

It is also important to note that the -xsd switch is complementary to the -xml switch when generating XML encoders and decoders. This is because the XML schema produced from the ASN.1 specification using the -xsd switch can be used to validate the XML messages generated using the XML encode functions. Similarly, an XML instance can be validated using the generated XML schema prior to decoding.

When C++ code generation is specified using the -xml switch, the generated *EncodeTo* and *DecodeFrom* methods in the PDU control class are set up to encode complete XML documents including the start document header as well as namespace attributes in the main element tag.

Generated Print Functions

The following options are available for generating code to print the contents of variables of generated types:

-print - This is the standard print option that causes print functions to be generated that output data to the standard output device (*stdout*).

-genPrtToStr - This option causes print functions to be generated that write their output to a memory buffer.

-genPrtToStrm - This option causes print functions to be generated that write their output to an output stream via a user-defined print callback function.

Print to Standard Output

The *-print* option causes functions to be generated that print the contents of variables of generated types to the standard output device. It is possible to specify the name of a .c or .cpp file as an argument to this option to specify the name of the file to which these functions will be written. This is an optional argument. If not specified, the functions are written to separate files for each module in the source file. The format of the name of each file is <*module*>*Print.c*. If an output filename is specified after the *–print* qualifier, all functions are written to that file.

The format of the name of each generated print function is as follows:

```
asn1Print_[<prefix>]<prodName>
```

where <prodName> is the name of the ASN.1 production for which the function is being generated and <prefix> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each generated print function is as follows:

```
asn1Print_<name> (ASN1ConstCharPtr name, <name>* pvalue)
```

In this definition, <name> denotes the prefixed production name defined above.

The name argument is used to hold the top-level name of the variable being printed. It is typically set to the same name as the *pvalue* argument in quotes (for example, to print an employee record, a call to *asn1Print_Employee ("employee", &employee)* might be used).

The *pvalue* argument is used to pass a pointer to a variable of the item to be printed.

If C++ code generation is specified, a *Print* method is added to the ASN1C control class for the type. This method takes only a *name* argument; the *pvalue* argument is obtained from the *msgData* reference contained within the class.

Print to String

The *-genPrtToStr* option causes functions to be generated that print the contents of variables of generated types to a given text buffer. This buffer can then be used to output the information to other mediums such as a file or window display.

It is possible to specify the name of a .c or .cpp file as an argument to this option to specify the name of the file to which these functions will be written. This is an optional argument. If not specified, the functions are written to separate files

for each module in the source file. The format of the name of each file is <*module*>*PrtToStr.c*. If an output filename is specified after the *–genPrtToStr* qualifier, all functions are written to that file.

The calling sequence for each generated print-to-string function is as follows:

```
asn1PrtToStr_<name> (ASN1ConstCharPtr name, <name>* pvalue,
                     char* buffer, int bufSize)
```

The *name* and *pvalue* arguments are the same as they were in the *-print* case.

The *buffer* and *bufSize* arguments are used to describe the memory buffer the text is to be written into. These arguments specify a fixed-size buffer. If the generated text is larger than the given buffer size, as much text as possible is written to the buffer and a –1 status value is returned. If the buffer is large enough to hold the text output, all text is written to the buffer and a zero status is returned.

For C++, a *toString* method is generated in the control class that calls the generated print-to-string function. In addition to the *name* argument, this method also takes a *buffer* and *bufSize* argument to describe the buffer to which the text is to be written.

Print to Stream

The *-genPrtToStrm* option causes functions to be generated that print the contents of variables of generated types to an output stream via a user-defined callback function. The advantage of these functions is that a user can register a callback function, and then the print stream is automatically directed to the callback function. This makes it easier to support print to file or print to window type of functionalities.

It is possible to specify the name of a .c or .cpp file as an argument to this option to specify the name of the file to which these functions will be written. This is an optional argument. If not specified, the functions are written to separate files for each module in the source file. The format of the name of each file is <*module*>*PrtToStrm.c*. If an output filename is specified after the *–genPrtToStrm* qualifier, all functions are written to that file.

Before calling generated print-to-stream functions, a callback function should be registered. Otherwise, a default callback function will be used that directs the print stream to the standard output device.

The callback function is declared as:

```
void (*rtPrintCallback)
   void* pPrntStrmInfo, ASN1ConstCharPtr fmtspec, va_list arglist);
```

The first parameter is user-defined data which will be passed to each invocation of the callback function. This parameter can be used to pass print stream specific data, for example, a file pointer if the callback function is to output data to a file. The second and third parameters to the callback function constitute the data to be printed, in the form of format specification followed by list of arguments. A simple callback function for printing to file can be defined as follows:

```
void writeToFile(void* pPrntStrmInfo, ASN1ConstCharPtr fmtspec, va_list arglist)
{
   FILE * fp = (FILE*) pPrntStrmInfo;
   vfprintf (fp, fmtspec, arglist);
}
```

Once the callback function is defined, it has to be registered with the runtime library. There are two types of registrations possible, global, which applies to all print streams and context level, which applies to print streams associated with a particular context.

For registering a global callback use:

```
rtSetGlobalPrintStream (rtPrintCallback myCallback, void * pStrmInfo);
```

For registering a context level callback use:

```
rtSetPrintStream (ASN1CTXT *pctxt,
   rtPrintCallback myCallback, void * pStrmInfo);
```

Once the callback function is registered, the calling of each generated print-to-stream function will result in output being directed to the callback function.

The print to stream functions are declared as follows:

```
    asn1PrtToStrm_<name>
       (ASN1CTXT *pctxt, ASN1ConstCharPtr name, <name>* pvalue);
```

The *name* and *pvalue* arguments are the same as they were in the *-print* case.

The *pctxt* argument is used to specify an ASN1C context. If a valid context argument is passed and there is a context level callback registered, then that callback will be used. If there is no context level callback registered, or the *pctxt* argument is NULL, then the global callback will be used. If there is no global callback registered, the default callback will be used which writes the print output to stdout.

If C++ code generation is specified, *setPrintStream* and *toStream* methods are added to the ASN1C control class for the type. The *setPrintStream* method takes only *myCallback* and *pStrmInfo* arguments; the *pctxt* argument is obtained from the context pointer reference contained within the class. The *toStream* method takes only a *name* argument; the *pctxt* argument is derived from the context pointer reference within the class and the *pvalue* argument is obtained from the *msgData* reference contained within the class.

Generated Compare Functions

The *–genCompare* option causes comparison functions to be generated. These functions can be used to compare the contents of two generated type variables.

If no output file is specified with the *–genCompare* qualifier, the functions are written to separate .c files for each module in the source file. The format of the name of each file is *<module>Compare.c*. If an output filename is specified after the *–genCompare* qualifier, all functions are written to that file.

The format of the name of each generated compare function is as follows:

```
asn1Compare_[<prefix>]<prodName>
```

where <prodName> is the name of the ASN.1 production for which the function is being generated and <prefix> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each generated compare function is as follows:

```
ASN1BOOL asn1Compare_<name> (ASN1ConstCharPtr name,
                             <name>* pvalue, <name>* pCmpValue,
                             char* errBuff, int errBufSize);
```

In this definition, <name> denotes the prefixed production name defined above.

The name argument is used to hold the top-level name of the variable being compared. It is typically set to the same name as the pvalue argument in quotes (for example, to compare employee records, a call to 'asn1Compare_Employee ("employee", &employee, etc.)' might be used).

The pvalue argument is used to pass a pointer to a variable of the item to the first item to be compared. The pCmpValue argument is used to pass the second value. The two items are then compared field-by-field for equality.

The errBuff and errBuffSize arguments are used to describe a text buffer into which information on what fields the comparison failed on is written. These arguments specify a fixed-size buffer – if the generated text is larger than the given buffer size, the text is terminated.

The return value of the function is a Boolean value that is true if the variables are equal and false if they are not.

Generated Copy Functions

The –genCopy option causes copy functions to be generated. These functions can be used to copy the content of one generated type variable to another.

If no output file is specified with the –genCopy qualifier, the functions are written to separate .c/.cpp files for each module in the source file. The format of the name of each file is <*module*>*Copy.c/.cpp* where <*module*> would be replaced with the name of the ASN.1 module. If an output filename is specified after the –genCopy qualifier, all functions are written to that file.

The format of the name of each generated copy function is as follows:

```
asn1Copy_[<prefix>]<prodName>
```

where <prodName> is the name of the ASN.1 production for which the function is being generated and <prefix> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each generated copy function is as follows:

```
void asn1Copy_<name> (ASN1CTXT* pctxt,
                      <name>* pSrcValue,
                      <name>* pDstValue);
```

In this definition, <name> denotes the prefixed production name defined above.

The pointer to the context structure (*pctxt*) provides a storage area for the function to store all variables that have been copied

The *pSrcValue* argument is used to pass a pointer to a variable to be copied. The *pDstValue* argument is used to pass the pointer to the destination value. The source value is then copied to the destination value field-by-field. Memory will be allocated for dynamic fields via calls to the *rtMemAlloc* function.

If C++ is used (-cpp option is specified) and PDU generation is not disabled (<noPDU> config option is not used) then the control class ASN1C_<name> additionally will contain:

- A copy constructor that can be used to create an exact copy of the class instance.

  The calling sequence is as follows:

  ```
  ASN1C_<name> (ASN1C_<name>& orginal);
  ```

  For example:

  ```
  ASN1C_PersonnelRecord (ASN1C_PersonnelRecord& original);
  ```

- A *getCopy* method that creates a copy of the *ASN1T_*<*name*> variable:

  ```
  ASN1T_<name>& getCopy (ASN1T_<name>* pDstData = 0);
  ```

For example:

```
ASN1T_PersonnelRecord& getCopy (ASN1T_PersonnelRecord* pDstData = 0);
```

The pDstData argument is used to pass the pointer to a destination variable where the value will be copied. It may be null, in this case the new ASN1T_<name> variable will be allocated via a call to the *rtMemAlloc* function.

- A *newCopy* method that will create a new, dynamically allocated copy of the referenced *ASN1T_* data member variable.

- An assignment operator. This is used to copy one instance of a control class to another one:

```
inline ASN1C_<name>& operator= (ASN1C_<name>& srcData)
{
   srcData.getCopy (&msgData);
   return *this;
}
```

For example:

```
inline ASN1C_PersonnelRecord& operator=
    (ASN1C_PersonnelRecord& srcData)
{
   srcData.getCopy (&msgData);
   return *this;
}
```

Finally, the class declaration might look as follows:

```
class EXTERN ASN1C_PersonnelRecord :
   public ASN1CType
{
protected:
   ASN1T_PersonnelRecord& msgData;
public:
   ASN1C_PersonnelRecord (
      ASN1MessageBuffer& msgBuf, ASN1T_PersonnelRecord& data);

   ASN1C_PersonnelRecord (ASN1C_PersonnelRecord& original);

      ...

   ASN1T_PersonnelRecord& getCopy (ASN1T_PersonnelRecord*
                                   pDstData = 0);

   ASN1T_PersonnelRecord* newCopy ();

   inline ASN1C_PersonnelRecord&
      operator= (ASN1C_PersonnelRecord& srcData)
```

```
        {
           srcData.getCopy (&msgData);
           return *this;
        }
     } ;
```

The generated *ASN1T<name>* structure will also contain an additional copy constructor if C++ is used and PDU generation is not disabled. A destructor is also generated if the type contains dynamic memory fields. This destructor will free the dynamic memory upon destruction of the type instance.

For example:

```
typedef struct EXTERN ASN1T_PersonnelRecord : public ASN1TPDU {
   ...
   ASN1T_PersonnelRecord () {
      m.uniPresent = 0;
      m.seqOfseqPresent = 0;
   }
   ASN1T_PersonnelRecord (ASN1C_PersonnelRecord& srcData);
   ~ASN1T_PersonnelRecord();
} ASN1T_PersonnelRecord;
```

This constructor is used to create an instance of the *ASN1T_<name>* type from an *ASN1C_<name>* control class object.

Memory Management of Copied Items

Prior to ASN1C version 5.6, dynamic memory of decoded or copied items would only be available as long as the control class instance and/or the message buffer object used to decode or copy the item remained in scope or was not deleted. This restriction no longer exists as long as the type being copied is a Protocol Data Unit (PDU). The copied item will now hold a reference to the context variable used to create the item and will increment the item's reference count. This reference is contained in the *ASN1TPDU* base class variable from which PDU data types are now derived. The dynamic memory within the item will persist until the item is deleted.

Generated Test Functions

The *–genTest* option causes test functions to be generated. These functions can be used to populate variables of generated types with random test data. The main purpose is to provide a code template to users for writing code to populate variables. This is quite useful to users because generated data types can become very complex as the ASN.1 schemas become more complex. It is sometimes difficult to figure out how to navigate all of the lists and pointers. Using *–genTest* can provide code that simply has to be modified to accomplish the population of a data variable with any type of data.

The generated test functions are written to a .c or .cpp file with a name of the following format:

```
<asn1ModuleName>Test.c
```

where <*asn1ModuleName*> is the name of the ASN.1 module that contains the type definitions.

The format of the name of each generated test function is as follows:

```
asn1Test_[<prefix>]<prodName>
```

where <prodName> is the name of the ASN.1 production for which the function is being generated and <prefix> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each generated test function is as follows:

```
<typeName>* pvalue = <testFunc> (ASN1CTXT* pctxt)
```

In this definition, <testFunc> denotes the formatted function name defined above.

The pctxt argument is used to hold a context pointer to keep track of dynamic memory allocation parameters. This is a basic "handle" variable that is used to make the function reentrant so that it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her program. The variable must have been previously initialized using the *rtInitContext* run-time function.

The pvalue argument is a pointer to hold the populated data variable. This variable is of the type generated for the ASN.1 production. The test function will automatically allocate dynamic memory using the run-time memory management for the main variable as well as variable length fields within the structure. This memory is tracked within the context structure and is released when the context structure is freed.

In the case of C++, a method is added to the generated control class for test code generation. The name of this method is **genTestInstance**. The prototype is as follows:

```
<typeName>* pvalue = <object>.genTestInstance();
```

where <typeName> is the ASN1T_<name> type name of the generated type and <object> is an instance of the ASN1C_<name> generated control class.

Generated Makefile

The *–genmake* option causes a portable makefile to be generated to assist in the C or C++ compilation of all of the generated C or C++ source files. This makefile contains a rule to invoke ASN1C to regenerate the .c and .h files if any of the dependent ASN.1 source files are modified. It also contains rules to compile all of the C or C++ source files. Header file dependencies are generated for all the C or C++ source files.

Two basic types of makefiles are generated:

1. A GNU compatible makefile. This makefile is compatible with the GNU make utility which is suitable for compiling code on Linux and many UNIX operating systems, and
2. A Microsoft Visual Studio compatible makefile. This makefile is compatible with the Microsoft Visual Studio *nmake* utility.

A GNU compatible makefile is produced by default, the Microsoft compatible file is produced when the *–w32* command line option is specified in addition to *–genmake.*

Both of these makefile types rely on definitions in the *platform.mk* make include file. This file contains parameters specific to different compiler and linker utilities available on different platforms. Typically, all the needs to be done to port to a different platform is to adjust the parameters in this file.

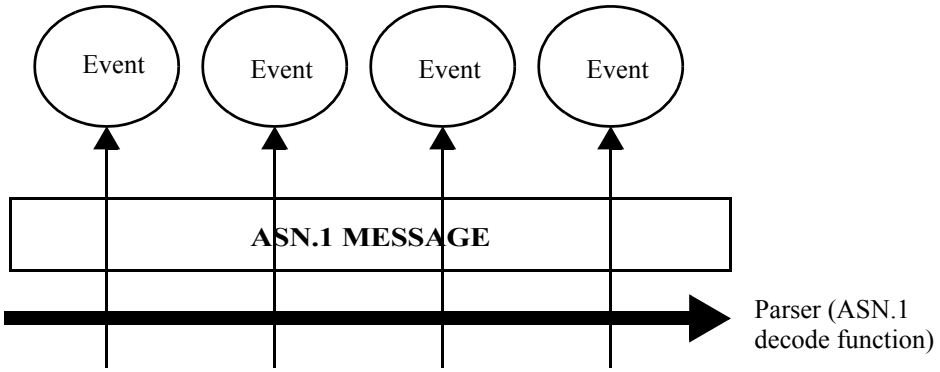
Event Handler Interface

The –events command line switch causes hooks for user-defined event handlers to be inserted into the generated decode functions. This feature is only available when C++ code generation is being done. What these event handlers do is up to the user. They fire when key message-processing events or errors occur during the course of parsing an ASN.1 message. They are similar in functionality to the Simple API for XML (SAX) that was introduced to provide a simple interface for parsing XML messages.

The -notypes option can be used in conjunction with the -events option to generate pure parsing functions. In this case, no C types or encode or decode functions are generated for the productions within the given ASN.1 source file. Instead, only a set of parser functions are generated that invoke the event handler callback functions. This gives the user total control over what is done with the message data. Data that is not needed can be discarded and only the parts of the message needed for a given application need to be saved.

How it Works

Users of XML parsers are probably already quite familiar with the concepts of SAX. Significant events are defined that occur during the parsing of a message. As a parser works through a message, these events are 'fired' as they occur by invoking user defined callback functions. These callback functions are also known as event handler functions. A diagram illustrating this parsing process is as follows:



The events are defined to be significant actions that occur during the parsing process. We will define the following events that will be passed to the user when an ASN.1 message is parsed:

1. **startElement** – This event occurs when the parser moves into a new element. For example, if we have a SEQUENCE { a, b, c } construct (type names omitted), this event will fire when we begin parsing a, b, and c. The name of the element is passed to the event handling callback function.

2. **endElement** – This event occurs when the parser leaves a given element space. Using the example above, these would occur after the parsing of a, b, and c are complete. The name of the element is once again passed to the event handling callback function.

3. **contents methods** – A series of virtual methods are defined to pass all of the different types of primitive values that might be encountered when parsing a message (see the event handler class definition below for a complete list).

4. **error** – This event will be fired when a parsing error occurs. It will provide fault-tolerance to the parsing process as it will give the user the opportunity to fix or ignore errors on the fly to allow the parsing process to continue.

These events are defined as unimplemented virtual methods in two base classes: *Asn1NamedEventHandler* (the first 3 events) and *Asn1ErrorHandler* (the error event). These classes are defined in the *asn1CppEvtHndlr.h* header file.

The start and end element methods are invoked when an element is parsed within a constructed type. The start method is invoked as soon as the tag/length is parsed in a BER message or the preamble/length is parsed in a PER message. The end method is invoked after the contents of the field are processed. The signature of these methods is as follows:

```
virtual void startElement (const char* name, int index) = 0;
virtual void endElement (const char* name, int index) = 0;
```

The *name* argument is used pass the element name. The *index* argument is used for SEQUENCE OF/SET OF constructs only. It is used to pass the index of the item in the array. This argument is set to –1 for all other constructs.

There is one contents method for passing each of the ASN.1 data types. Some methods are used to handle several different types. For example, the *charValue* method is used for values of all of the different character string types (IA5String, NumericString, PrintableString, etc.) as well as for big integer values. Note that this method is overloaded. The second implementation is for 16-bit character strings. These strings are represented as an array of unsigned short integers in ASN1C. All of the other contents methods correspond to a single equivalent ASN.1 primitive type.

The error handler base class has a single virtual method that must be implemented. This is the error method and this has the following signature:

```
virtual int error (ASN1CTXT* pCtxt, ASN1CCB* pCCB, int stat) = 0;
```

In this definition, pCtxt is a pointer to the standard ASN.1 context block that should already be familiar. The pCCB structure is known as a "Context Control Block". This can be thought of as a sub-context used to control the parsing of nested constructed types within a message. It is included as a parameter to the error method mainly to allow access to the "seqx" field. This is the sequence element index used when parsing a SEQUENCE construct. If parsing a particular element is to be retried, this item must be decremented within the error handler.

How to Use It

To define event handlers, two things must be done:

1. One or more new classes must be derived from the *Asn1NamedEventHandler* and/or the *Asn1ErrorHandler* base classes. All pure virtual methods must be implemented.

2. Objects of these classes must be created and registered prior to calling the generated decode method or function.

The best way to illustrate this procedure is through examples. We will first show a simple event handler application to provide a customized formatted printout of the fields in a PER message. Then we will show a simple error handler that will ignore unrecognized fields in a BER message.

Example 1: A Formatted Print Handler

The ASN1C evaluation and distribution kits include a sample program for doing a formatted print of parsed data. This code can be found in the *cpp/sample_per/eventHandler* directory. Parts of the code will be reproduced here for reference, but refer to this directory to see the full implementation.

The format for the printout will be simple. Each element name will be printed followed by an equal sign (=) and an open brace ({) and newline. The value will then be printed followed by another newline. Finally, a closing brace (}) followed by another newline will terminate the printing of the element. An indentation count will be maintained to allow for a properly indented printout.

A header file must first be created to hold our print handler class definition (or the definition could be added to an existing header file). This file will contain a class derived from the *Asn1NamedEventHandler* base class as follows:

```
class PrintHandler : public Asn1NamedEventHandler {
 protected:
   const char* mVarName;
   int mIndentSpaces;
 public:
   PrintHandler (const char* varName);
   ~PrintHandler ();
   void indent ();
   virtual void startElement (const char* name, int index = -1);
   virtual void endElement (const char* name, int index = -1);
   virtual void boolValue (ASN1BOOL value);

   ... other virtual contents method declarations

}
```

In this definition, we chose to add the mVarName and mIndentSpaces member variables to keep track of these items. The user is free to add any type of member variables he or she wants. The only firm requirement in defining this derived class is the implementation of the virtual methods.

We implement these virtual methods as follows:

In *startElement*, we print the name, equal sign, and opening brace:

```
void PrintHandler::startElement (const char* name, int index)
{
   indent();
   printf ("%s = {\n", name);
   mIndentLevel++;
}
```

In this simplified implementation, we simply indent (this is another private method within the class) and print out the name, equal sign, and opening brace. We then increment the indent level. Note that this is a highly simplified form. We don't even bother to check if the index argument is greater than or equal to zero. This would determine if a '[x]' should be appended to the element name. In the sample program that is included with the compiler distribution, the implementation is complete.

In *endElement*, we simply terminate our brace block as follows:

```
void PrintHandler::endElement (const char* name, int index)
{
   mIndentLevel--;
   indent();
```

```
   printf ("}\n");
}
```

All that each of the various value methods have to do is print a stringified representation of the value out to stdout. For example, the *intValue* callback would just print an integer value:

```
void PrintHandler::intValue (int value)
{
   indent();
   printf ("%d\n", value);
}
```

Next, we need to create an object of our derived class and register it prior to invoking the decode method. In the *reader.cpp* program, the following lines do this:

```
   // Create and register an event handler object

   PrintHandler* pHandler = new PrintHandler ("employee");
   decodeBuffer.addEventHandler (pHandler);
```

The *addEventHandler* method defined in the *Asn1MessageBuffer* base class is the mechanism used to do this. Note that event handler objects can be stacked. Several can be registered before invoking the decode function. When this is done, the entire list of event handler objects is iterated through and the appropriate event handling callback function invoked whenever a defined event is encountered.

The implementation is now complete. The program can now be compiled and run. When this is done, the resulting output is as follows:

```
employee = {
   name = {
      givenName = {
         "John"
      }
      initial = {
         "P"
      }
      familyName = {
         "Smith"
      }
   }
   ...
```

This can certainly be improved. For one thing it can be changed to print primitive values out in a "name = value" format (i.e., without the braces). But this should provide the general idea of how it is done.

Example 2: An Error Handler

Despite the addition of things like extensibility and version brackets, ASN.1 implementations get out-of-sync. For situations such as this, the user needs some way to intervene in the parsing process to set things straight. This is fault-tolerance – the ability to recover from certain types of errors.

The error handler interface is provided for this purpose. The concept is simple. Instead of throwing an exception and immediately terminating the parsing process, a user defined callback function is first invoked to allow the user to check the error. If the user can fix the error, all he or she needs to do is apply the appropriate patch and return a status of ASN_OK. The parser will be none the wiser. It will continue on thinking everything is fine.

This interface is probably best suited for recovering from errors in BER or DER instead of PER. The reason is the TLV format of BER makes it relatively easy to skip an element and continue on. It is much more difficult to find these boundaries in PER.

Our example can be found in the *cpp/sample_ber/errorHandler* subdirectory. In this example, we have purposely added a bogus element to one of the constructs within an encoded employee record. The error handler will be invoked when this element is encountered. Our recovery action will simply be to print out a warning message, skip the element, and continue.

As before, the first step is to create a class derived from the *Asn1ErrorHandler* base class. This class is as follows:

```
class MyErrorHandler : public Asn1ErrorHandler {
 public:

   // The error handler callback method.  This is the method
   // that the user must override to provide customized
   // error handling..

   virtual int error (ASN1CTXT* pCtxt, ASN1CCB* pCCB, int stat);

} ;
```

Simple enough. All we are doing is providing an implementation of the error method.

Implementing the error method requires some knowledge of the run-time internals. In most cases, it will be necessary to somehow alter the decoding buffer pointer so that the same field isn't looked at again. If this isn't done, an infinite loop can occur as the parser encounter the same error condition over and over again. The run-time functions *xd_NextElement* or *xd_OpenType* might be useful in the endeavor as they provide a way to skip the current element and move on to the next item.

Our sample handler corrects the error in which an unknown element is encountered within a SET construct. This will cause the error status ASN_E_NOTINSET to be generated. When the error handler sees this status, it prints information on the error that was encountered to the console, skips to the next element, and then returns an ASN_OK status that allows the decoder to continue. If some other error occurred (i.e., status was not equal to ASN_E_NOTINSET), then the original status is passed out which forces the termination of the decoding process.

The full text of the handler is as follows:

```
int MyErrorHandler::error (ASN1CTXT* pCtxt, ASN1CCB* pCCB, int stat)
{
   // This handler is set up to look explicitly for ASN_E_NOTINSET
   // errors because we know the SET might contain some bogus elements..

   if (stat == ASN_E_NOTINSET) {
```

```
      // Print information on the error that was encountered

      printf ("decode error detected:\n");
      xu_perror (pCtxt);
      printf ("\n");

      // Skip element

      xd_NextElement (pCtxt);

      // Return an OK status to indicate parsing can continue

      return ASN_OK;
   }

   else return stat;  // pass existing status back out

}
```

Now we need to register the handler. Unlike event handlers, only a single error handler can be registered. The method to do this in the message buffer class is *setErrorHandler*. The following two lines of code in the reader program register the handler:

```
   MyErrorHandler errorHandler;

   decodeBuffer.setErrorHandler (&errorHandler);
```

The error handlers can be as complicated as you need them to be. You can use them in conjunction with event handlers in order to figure out where you are within a message in order to look for a specific error at a specific place. Or you can be very generic and try to continue no matter what.

Example 3: A Pure Parser to Convert PER-encoded Data to XML

A pure-parser is created by using the -notypes option along with the -events option. In this case, no backing data types to hold decoded data are generated. Instead, parsing functions are generated that store the data internally within local variables inside the parsing functions. This data is dispatched to the callback functions and immeditely disposed of upon return from the function. It is up to the user to decide inside the callback handler what they want to keep and they must make copies at that time. The result is a very fast and low-memory consuming parser that is ideal for parsing messages in which only select parts of the messages are of interest.

Another use case for pure-parser functions is validation. These functions can be used to determine if a PER message is valid without going through the high overhead operation of decoding. They can be used on the front-end of an application to reject invalid messages before processing of the messages is done. In some cases, this can result in significantly increased performance.

An example of a pure-parser can be found in the *cpp/sample_per/per2xmlEH* directory. This program uses a pure-parser to convert PER-encoded data into XML. The steps in creating an event handler are the same as in Example 1 above. An implementation of the *Asn1NamedEventHandler* interface must be created. This is done in the *xmlHandler.h* and

xmlHandler.cpp files. A detailed discussion of this code will not be provided here. What it does in a nutshell is adds the angle brackets (< >) around the element names in the *startElement* and *endElement* callbacks. The data callbacks simple output a textual representation of the data as they do in the print handler case.

The only difference in *reader.cpp* from the other examples is that:

1. There is no declaration of an employee variable to hold decoded data because no type for this variable was generated, and

2. The *Parse* method is invoked instead of the *Decode* method. This is the generated method definition for a pure-parser. If one examines the generated class definitions, they will see that no *Encode* or *Decode* methods were generated.

Compiling and running this program will show the encoded PER message written to stdout as an XML message. The resulting message is also saved in the *message.xml* file.

IMPORT/EXPORT of Types

ASN1C allows productions to be shared between different modules through the ASN.1 IMPORT/EXPORT mechanism. The compiler parses but ignores the EXPORTS declaration within a module. As far as it is concerned, any type defined within a module is available for import by another module.

When ASN1C sees an IMPORT statement, it first checks its list of loaded modules to see if the module has already been loaded into memory. If not, it will attempt to find and parse another source file containing the module. The logic for locating the source file is as follows:

1. The configuration file (if specified) is checked for a <sourceFile> element containing the name of the source file for the module.

2. If this element is not present, the compiler looks for a file with the name <ModuleName>.asn where module name is the name of the module specified in the IMPORT statement.

In both cases, the –I command line option can be used to tell the compiler where to look for the files.

The other way of specifying multiple modules is to include them all within a single ASN.1 source file. It is possible to have an ASN.1 source file containing multiple module definitions in which modules IMPORT definitions from other modules. An example of this would be the following:

```
ModuleA DEFINITIONS ::= BEGIN
   IMPORTS B From ModuleB;

   A ::= B

END

ModuleB DEFINITIONS ::= BEGIN

   B ::= INTEGER

END
```

This entire fragment of code would be present in a single ASN.1 source file.

ASN.1 to XML Schema Conversion

The ASN1C compiler contains the capability of generating corresponding XML Schema type definitions from ASN.1 types. This capability is also present in a free online tool (ASN2XSD) that may be accessed via the following URL:

http://www.obj-sys.com/asn2xsdform.shtml

ASN.1 types are converted to XML Schema and ASN.1 values are converted to XML. ASN.1 value sets, which are essentially a set of constraints, get converted to facets in XML Schema. ASN.1 Class, Object and ObjectSet conversions are not supported at this time.

Mapping of Top-Level Constructs

An ASN.1 module name is mapped to an XML schema namespace. ASN.1 IMPORT statements are mapped to XSD *import* statements. The ASN.1 EXPORT statement does not have a corresponding construct in XSD.

The general form of the XSD namespace and import statements would be as follows:

```
<?xml version="1.0"?>
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="<URL>/<ModuleName>"

   <!-- following line would be added for imported module namespace -->
   xmlns:<ImportedModuleName>="<importURL>/<ImportedModuleName>"
      elementFormDefault="qualified">

   <xsd:import namespace="<importURL>/<ImportedModuleName>"
      schemaLocation="<ImportedModuleName>.xsd"/>
```

In this definition, the items in blue (or *italics*) would be replaced with text from the ASN.1 specification being converted or a configuration file. The <*ModuleName*> and <*ImportedModuleName*> items would come from the ASN.1 specification. The <*URL*> and <*importURL*> would be configuration file parameters.

Mapping of ASN.1 Types

Each ASN.1 type is mapped to a corresponding XSD type. Some ASN.1 types have a natural mapping to an XSD type (for example, an ASN.1 BOOLEAN type maps to an xsd:boolean type). In other cases, custom types were needed because a natural mapping did not exist within XSD (for example, there was no direct mapping for an ASN.1 BIT-STRING type). These custom types can be found in the low-level ASN.1 XML schema definitions library at the following URL:

http://www.obj-sys.com/v1.0/XMLSchema/asn1.xsd

The following sections describe the mappings for each of the ASN.1 built-in types.

BOOLEAN

The ASN.1 BOOLEAN type is mapped to the XSD *boolean* built-in type.

ASN.1 production:

```
TypeName ::= BOOLEAN
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:boolean"/>
</xsd:simpleType>
```

INTEGER

The ASN.1 INTEGER type is converted into one of several XSD built-in types depending on value range constraints on the integer type definition.

The default conversion if the INTEGER value contains no constraints is to the XSD *integer* type:

ASN.1 production:

```
TypeName ::= INTEGER
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:integer"/>
</xsd:simpleType>
```

If the integer has a value range constraint that allows a more restrictive XSD type to be used, then that type will be used. For example, if a range of 0 to 255 (inclusive) is specified, an XSD *unsignedByte* would be used because it maps exactly to this range. The following table shows the range values for each of the INTEGER type mappings:

Lower Bound	Upper Bound	XSD Type
-128	127	byte
0	255	unsignedByte
-32768	32767	short
0	65535	unsignedShort
-2147483648	2147483647	integer
0	4294967295	unsignedInt
-9223372036854775808	9223372036854775807	long
0	18446744073709551615	unsignedLong

Ranges beyond “long” or “unsignedLong” will cause the integer value to be treated as a “big integer”. This will map to an *xsd:string* type. An integer can also be specified to be a big integer using the ASN1C <isBigInteger/> configuration file setting.

If constraints are present on the INTEGER type that are not exactly equal to the lower and upper bounds specified above, then *xsd:minInclusive* and *xsd:maxInclusive* facets will be added to the XSD type mapping. For example, the mapping of "I ::= INTEGER (0..10)" would be done as follows:

1. The most restrictive type would first be chosen based on the constraints. In this case, *xsd:byte* would be used because it appears first on the list above.

2. Then the *xsd:minInclusive* and *xsd:maxInclusive* facets would be added to further restrict the type.

This would result in the following mapping:

```
<xsd:simpleType name="I">
   <xsd:restriction base="xsd:byte">
      <xsd:minInclusive value="0">
      <xsd:maxInclusive value="10">
   </xsd:restriction>
</xsd:simpleType>
```

BIT STRING

There is no built-in XSD type that corresponds to the ASN.1 BIT STRING type. For this reason, a custom type was created in the ASN2XSD run-time library (*asn1.xsd*) to model this type. This type is *asn1:BitString* and has the following definition:

```
<xsd:simpleType name="BitString">
   <xsd:restriction base="xsd:token">
      <xsd:pattern value="[0-1]{0,}"/>
   </xsd:restriction>
</xsd:simpleType>
```

The ASN.1 BIT STRING type is converted into a reference to this custom type as follows:

ASN.1 production:

```
TypeName ::= BIT STRING
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="asn1:BitString"/>
</xsd:simpleType>
```

Sized BIT STRING

The ASN.1 BIT STRING type may contain a size constraint. This is converted into *minLength* and *maxLength* facets in the generated XSD definition:

ASN.1 production:

```
TypeName ::= BIT STRING (SIZE (lower..upper))
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="asn1:BitString">
      <xsd:minLength value="lower"/>
      <xsd:maxLength value="upper"/>
   </xsd:restriction>
</xsd:simpleType>
```

BIT STRING with Named Bits

A bit string with named bits is handled differently than a normal bit string. This is because the primary use of named bits it to define a bit map of selected bit items. For this reason, a list of enumerated items is used for the type. This allows the names of the bits to be specified in an XML instance of the type. The type also contains application information in the form of an annotation that allows an application to map the items specified in a list to binary bits in a bitmap.

The formal mapping of an ASN.1 BIT STRING with named bits to XSD is as follows:

ASN.1 production:

```
TypeName ::= BIT STRING { b1(n1), b2(n2) }
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
  <xsd:union memberTypes="asn1:BitString">
    <xsd:list>
      <xsd:simpleType>
        <xsd:restriction base="xsd:token">
          <xsd:enumeration value="b1" asn1:bitno="n1"/>
          <xsd:enumeration value="b2" asn1:bitno="n2"/>
          ...
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:list>
  </xsd:union>
</xsd:simpleType>
```

Previous versions of ASN1C generate annotation code for this definition. This can still be generated with current version using the *-appinfo* option.

Generated XSD code with annotation:

```
<xsd:simpleType name="TypeName">
  <xsd:annotation>
    <xsd:appinfo>
```

```
      <asn1:NamedBitInfo>
        <asn1:NamedBit name="b1" bitNumber="n1">
        <asn1:NamedBit name="b2" bitNumber="n2">
        ...
      </asn1:NamedBitInfo>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:union memberTypes="asn1:BitString">
    <xsd:list>
      <xsd:simpleType>
        <xsd:restriction base="xsd:token">
          <xsd:enumeration value="b1"/>
          <xsd:enumeration value="b2"/>
          ...
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:list>
  </xsd:union>
</xsd:simpleType>
```

The appinfo will not be needed if the bits are sequentially numbered starting at zero. The compiler will be able to figure out the bit numbers based on their position in the document.

Example

The following ASN.1 BIT STRING type:

```
ColorSet ::= BIT STRING (blue(1), green(3), red(5))
```

maps to the following XSD type

```
<xsd:simpleType name="ColorSet">
  <xsd:union memberTypes="asn1:BitString">
    <xsd:list>
      <xsd:simpleType>
        <xsd:restriction base="xsd:token">
          <xsd:enumeration value="blue" asn1:bitno="1"/>
          <xsd:enumeration value="green" asn1:bitno="3"/>
          <xsd:enumeration value="red" asn1:bitno="5"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:list>
  </xsd:union>
</xsd:simpleType>
```

OCTET STRING

The ASN.1 OCTET STRING type is converted into the XSD *hexBinary* type.

ASN.1 production :

```
TypeName ::= OCTET STRING
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:hexBinary"/>
</xsd:simpleType>
```

Sized OCTET STRING

The ASN.1 OCTET STRING type may contain a size constraint. This is converted into *minLength* and *maxLength* facets in the generated XSD definition:

ASN.1 production:

```
TypeName ::= OCTET STRING (SIZE (lower..upper))
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="hexBinary">
      <xsd:minLength value="lower"/>
      <xsd:maxLength value="upper"/>
   </xsd:restriction>
</xsd:simpleType>
```

Character String Types

All ASN.1 character string useful types (*IA5String*, *VisibleString*, etc.) are mapped to the XSD *string* type.

ASN.1 production:

```
TypeName ::= ASN1CharStringType
```

in this definition, *ASN1CharStringType* would be replaced with one of the ASN.1 Character String types such as *VisibleString*.

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="string"/>
</xsd:simpleType>
```

ASN.1 character string types may contain a size constraint. This is converted into *minLength* and *maxLength* facets in the generated XSD definition:

ASN.1 production:

```
TypeName ::= ASN1CharStringType (SIZE (lower..upper))
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:string">
      <xsd:minLength value="lower"/>
      <xsd:maxLength value="upper"/>
   </xsd:restriction>
</xsd:simpleType>
```

ASN.1 character string types may also contain permitted alphabet or pattern constraints. These are converted into *pattern* facets in the generated XSD definition:

ASN.1 production:

```
TypeName ::= ASN1CharStringType (FROM (charSet))
```

or

```
TypeName ::= ASN1CharStringType (PATTERN (pattern))
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="pattern"/>
      <xsd:maxLength value="upper"/>
   </xsd:restriction>
</xsd:simpleType>
```

In this case, the permitted alphabet character set (*charSet*) is converted into a corresponding *pattern* for use in the generated XML schema definition.

Time String Types

The ASN.1 *GeneralizedTime* and *UTCTime* types are mapped to the XSD *dateTime* type.

ASN.1 production:

```
TypeName ::= ASN1TimeStringType
```

in this definition, *ASN1TimeStringType* would be replaced with either *GeneralizedTime* or *UTCTime*.

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="dateTime"/>
</xsd:simpleType>
```

ENUMERATED

The ASN.1 ENUMERATED type is converted into an XSD token type with enumeration items. The enumeration items correspond to the enumerated identifiers in the type. If the enumerated items contain numbers (i.e do not follow the standards sequence), then an *asn1:value* attribute is added to the type to allow an application to map the enumerated identifiers to numbers. If an *asn1:value* attribute is not present, then an application can safely assume that the enumerated identifiers are in sequential order starting at zero.

ASN.1 production:

```
TypeName ::= ENUMERATED (id1(val1), id2(val2), etc.)
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:token">
      <xsd:enumeration name="id1" asn1:value="val1">
      <xsd:enumeration name="id2" asn1:value="val2">
   </xsd:restriction>
</xsd:simpleType>
```

Previous versions of ASN1C generate annotation code for this definition. This can be generated with current version using the *-appinfo* option.

Generated XSD code with annotation:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:token">
      <xsd:enumeration name="id1">
      <xsd:enumeration name="id2">
   </xsd:restriction>
   <xsd:annotation>
      <xsd:appinfo>
         <asn1:EnumInfo>
           <asn1:EnumItem name="id1" value="val1"/>
           <asn1:EnumItem name="id2" value="val2"/>
         </asn1:EnumInfo>
      </xsd:appinfo>
   </xsd:annotation>
</xsd:simpleType>
```

Example

The following ASN.1 enumerated type:

```
Colors ::= ENUMERATED (blue(1), green(3), red(5))
```

maps to the following XSD type

```
<xsd:simpleType name="Colors">
   <xsd:restriction base="xsd:token">
```

```
            <xsd:enumeration name="blue" asn1:value="1">
            <xsd:enumeration name="green" asn1:value="3">
            <xsd:enumeration name="red" asn1:value="5">
         </xsd:restriction>
      </xsd:simpleType>
```

Note that if the identifiers in the enumerated type did not contain numbers (i.e. if the type was 'ENUMERATED (blue, green, red)'), then the annotation would not be necessary on the type above.

Extensible ENUMERATED

An ENUMERATED type may include an extensibility marker (…) to indicate that enumerated items not defined in the current set may be added at a later time. The generated XSD type in this case must allow items outside of the base set to not cause a validation error. This is accomplished by adding a union with an *xsd:token* type to the generated type definition:

ASN.1 production:

```
TypeName ::= ENUMERATED (id1(val1), id2(val2), etc, ...)
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:union memberTypes="xsd:token">
      <xsd:simpleType>
         <xsd:restriction base="xsd:token">
            <xsd:enumeration name="id1" asn1:value="val1">
            <xsd:enumeration name="id2" asn1:value="val2">
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:union>
</xsd:simpleType>
```

NULL

The ASN.1 NULL type does not generate an associated XSD type definition.

OBJECT IDENTIFIER

There is no built-in XSD type that corresponds to the ASN.1 OBJECT IDENTIFIER type. For this reason, a custom type was created in the Objective Systems XSD run-time library (*osxsdlib.xsd*) to model this type. This type is *asn1:OBJECT-IDENTIFIER* and has the following definition:

```
<xsd:simpleType name="OBJECT-IDENTIFIER">
   <xsd:restriction base="xsd:token">
      <xsd:pattern value=
         "[0-2]((\.[1-3]?[0-9])(?\.\d)*)?"/>
   </xsd:restriction>
```

```
</xsd:simpleType>
```

The pattern enforces the rule in the X.680 standard that the first arc value of an OID must be between 0 and 2, the second arc must be between 0 and 39, and the remaining arcs can be any number. The ASN.1 OBJECT IDENTIFIER type is converted into a reference to this custom type as follows:

ASN.1 production:

```
TypeName ::= OBJECT IDENTIFIER
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="asn1:OBJECT-IDENTIFIER"/>
</xsd:simpleType>
```

RELATIVE-OID

There is no built-in XSD type that corresponds to the ASN.1 RELATIVE-OID type. For this reason, a custom type was created in the Objective Systems XSD run-time library (*osxsdlib.xsd*) to model this type. This type is *asn1:RELATIVE-OID* and has the following definition:

```
<xsd:simpleType name="RELATIVE-OID">
   <xsd:restriction base="xsd:token">
      <xsd:pattern value="\d(\.\d)*"/>
   </xsd:restriction>
</xsd:simpleType>
```

This is similar to the OBJECT IDENTIFIER type discussed in the previous section except in this case, the patter in simpler. The arc numbers in a RELATIVE-OID are not restricted in any way, hence the simpler pattern. The ASN.1 RELATIVE-OID type is converted into a reference to this custom type as follows:

ASN.1 production:

```
TypeName ::= RELATIVE-OID
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
  <xsd:restriction base="asn1:RELATIVE-OID"/>
</xsd:simpleType>
```

REAL

The ASN.1 REAL type is mapped to the XSD *double* built-in type.

ASN.1 production:

```
TypeName ::= REAL
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:double"/>
</xsd:simpleType>
```

SEQUENCE

An ASN.1 SEQUENCE is a constructed type consisting of a series of element definitions that must appear in the specified order. This is very similar to the XSD *sequence* complex type and is therefore mapped to this type.

The basic mapping is as follows:

ASN.1 production:

```
TypeName ::= SEQUENCE {
   element1-name element1-type,
   element2-name element2-type,
      ...
}
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:sequence>
      <xsd:element name="element1-name" type="element1-type"/>
      <xsd:element name="element2-name" type="element2-name"/>
      …
   </xsd:sequence>
</xsd:complexType>
```

OPTIONAL keyword

Elements within a sequence can be declared to be optional using the OPTIONAL keyword. This indicates that the element is not required in the encoded message. XSD contains the *minOccurs* facet that can be used to model this behavior. Setting *minOccurs* equal to zero is the equivalent to declaring an element to be optional because this indicates the element can appear zero to one times in the definition.

For example, the following ASN.1 SEQUENCE type:

```
OptInt ::= SEQUENCE {
   anIntINTEGER OPTIONAL
}
```

will cause the following XSD complex type to be generated:

```
<xsd:complexType name=”OptInt”>
  <xsd:sequence>
    <xsd:element name=”anInt” type=”xsd:integer” minOccurs=”0”/>
```

```
  </xsd:sequence>
</xsd:complexType>
```

DEFAULT keyword

The DEFAULT keyword allows a default value to be specified for elements within the SEQUENCE. XSD contains a *default* facet that can be used to map elements with this keyword.

For example, the following ASN.1 SEQUENCE type:

```
DfltInt ::= SEQUENCE {
   anIntINTEGER DEFAULT 1
}
```

will cause the following XSD complex type to be generated:

```
<xsd:complexType name=”DfltInt”>
  <xsd:sequence>
    <xsd:element name=”anInt” type=”xsd:integer” default=”1”/>
  </xsd:sequence>
</xsd:complexType>
```

Note that in XSD, default values can only be specified for simple (primitive) types. ASN.1 allows for the specification of default values on complex (constructed) types as well. If an ASN.1 type is encountered that contains a complex default value, the value is dropped in the conversion to XSD.

Extension Elements

If the SEQUENCE type is extensible (i.e., contains an ellipses marker …), a special element will be inserted to allow unknown elements to be validated. This special element is as follows:

```
<xsd:any namespace="##other" processContents="lax"/>
```

This element declaration allows any additional elements from other namespaces to exist in a message instance without causing a validation or decoding error. Note the restriction that the element must be defined in a different namespace. This is necessary because if the element existed in the same namespace as other elements, the content model would be non-deterministic. The reason is because a validation program would not be able to determine if the last element is a sequence was a defined element or an extension element.

The extension element is marked with non-native attribute description. For example:

```
DfltInt ::= SEQUENCE {
   anInt   INTEGER,
   ...,
   extElm  BOOLEAN
}

Generated XSD code:

<xsd:complexType name="DfltInt">
   <xsd:sequence>
```

```
        <xsd:element name="anInt" type="xsd:integer"/>
        <xsd:element name="extElm" minOccurs="0" type="xsd:boolean"
          asn1:description="extension element"/>
        <xsd:any namespace="##other" processContents="lax" minOccurs="0"/>
      </xsd:sequence>
   </xsd:complexType>
```

Note: In the ASN.1 SEQUENCE or SET type, all extension elements are optional, whether marked that way or not.

SET

An ASN.1 SET is a constructed type consisting of a series of element definitions that must appear in any order. This is very similar to the XSD *all* complex type and is therefore mapped to this type.

The basic mapping is as follows:

ASN.1 production:

```
TypeName ::= SET {
   element1-name element1-type,
   element2-name element2-type,
      ...
   }
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:all>
      <xsd:element name="element1-name" type="element1-type"/>
      <xsd:element name="element2-name" type="element2-name"/>
      …
   </xsd:all>
</xsd:complexType>
```

The rules for mapping elements with optional and default values to XSD that were described in the SEQUENCE section above are also applicable to the SET type.

SEQUENCE OF / SET OF

The ASN.1 SEQUENCE OF or SET OF type is used to specify a repeating collection of a given element type. This is similar to an array type in a high-level programming language. For all practical purposes, SEQUENCE OF and SET OF are identical. The remainder of this section will refer to the SEQUENCE OF type only. It can be assumed that all of the defined mappings apply to the SET OF type as well.

The way the SEQUENCE OF type is mapped to XSD depends on the type of the referenced element. If the type is one of the following ASN.1 primitive types (or a type reference that references one of these types):

- BOOLEAN
- INTEGER
- ENUMERATED
- REAL

The mapping is to the XSD *list* type. This is a list of space-separated identifiers. The syntax is as follows:

ASN.1 production:

```
TypeName ::= SEQUENCE OF ElementType
```

Generated XSD code:

```
<xsd:simpleType name="TypeName">
   <xsd:list itemType="ElementType">
</xsd:simpleType>
```

This will be referred to as the simple case from this point forward.

If the element type is any other type than those listed above, the ASN.1 type is mapped to an XSD sequence complex type that contains a single element of the element type. The generated XSD type also contains the *maxOccurs* (and possibly the *minOccurs*) facet to specify the array bounds.

The general mapping of an unbounded SEQUENCE OF type (i.e. one with no size constraint) to XSD is as follows:

ASN.1 production:

```
TypeName ::= SEQUENCE OF ElementType
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:sequence>
      <xsd:element name="ElementType" type="ElementType"
       maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```

In this definition, the element type name is the name of the ASN.1 element type. The element type in the XSD definition is the equivalent XSD type for the ASN.1 element type.

As of the 2002 version of the ASN.1 standards, it is now possible to include an element identifier name before the element type name in a SEQUENCE OF definition. This makes it possible to control the name of the element used in the generated XSD definition. The mapping for this case is as follows:

ASN.1 production:

```
TypeName ::= SEQUENCE OF elementName ElementType
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:sequence>
      <xsd:element name="elementName" type="ElementType"
       maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```

Example

The following shows the mapping for a SEQUENCE OF INTEGER. Since INTEGER is one of the simple types listed above, an XSD *list* type is used:

ASN.1 production:

```
SeqOfInt ::= SEQUENCE OF INTEGER
```

Generated XSD code:

```
<xsd:complexType name="SeqOfInt">
   <xsd:list itemType="xsd:integer">
</xsd:complexType>
```

The following shows the mapping for a SEQUENCE OF UTF8String. Since UTF8String is not one of the simple types listed above, an XSD *sequence* type is used:

ASN.1 production:

```
SeqOfUTF8 ::= SEQUENCE OF UTF8String
```

Generated XSD code:

```
<xsd:complexType name="SeqOfUTF8">
   <xsd:sequence>
      <xsd:element name="UTF8String" type="xsd:string"
       maxOccurs=”unbounded”/>
  </xsd:sequence>
</xsd:complexType>
```

Note that the element name is the name of the element type. To change the element name, the ASN.1 form that allows an element name could be used:

ASN.1 production:

```
SeqOfUTF8 ::= SEQUENCE OF myString UTF8String
```

Generated XSD code:

```
<xsd:complexType name="SeqOfUTF8">
   <xsd:sequence>
      <xsd:element name="myString" type="xsd:string"
       maxOccurs=”unbounded”/>
   </xsd:sequence>
</xsd:complexType>
```

Sized SEQUENCE OF / SET OF

The SEQUENCE OF type may contain a size constraint. If this is the case, the XSD *minOccurs* and *maxOccurs* facets are used to constrain the value to the given size.

ASN.1 production:

```
TypeName ::= SEQUENCE (SIZE (lower..upper)) OF ElementType
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:sequence>
      <xsd:element name="ElementType" type="ElementType"
      minOccurs="lower" maxOccurs="upper"/>
   </xsd:sequence>
</xsd:complexType>
```

This mapping is for the complex case. For the simple case (i.e XSD *list* case), the XSD *minLength* and/or *maxLength* facets are used to constraint the length:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction>
     <xsd:simpleType>
        <xsd:list itemType="ElementType">
     </xsd:simpleType>
     <xsd:minLength value="lower"/>
     <xsd:maxLength value="upper"/>
   </xsd:restriction>
</xsd:simpleType>
```

CHOICE

The ASN.1 CHOICE type is used to specify a list of alternative elements from which a single element can be selected. This type is mapped to the XSD *choice* complex type. The mapping is as follows:

ASN.1 production:

```
TypeName ::= CHOICE {
   element1-name element1-type,
   element2-name element2-type,
    ...
   }
```

Generated XSD code:

```
<xsd:complexType name="TypeName">
   <xsd:choice>
      <xsd:element name="element1-name" type="element1-type"/>
      <xsd:element name="element2-name" type="element2-name"/>
      …
   </xsd:choice>
</xsd:complexType>typedef struct {
```

This is similar to the SEQUENCE and SET cases described above. The only difference is that *xsd:choice* is used instead of *xsd:sequence* or *xsd:all*.

The CHOICE type cannot have elements marked as optional (OPTIONAL) or elements that contain default values (DEFAULT) as was the case for SEQUENCE and SET.

If the CHOICE type is extensible (i.e., contains an ellipses marker …), a special element will be inserted to allow unknown alternative to be validated. This element is as follows:

```
<xsd:any namespace="##other" processContents="lax"/>
```

This element declaration allows any additional elements from other namespaces to exist in a message instance without causing a validation or decoding error. Note the restriction that the element must be defined in a different namespace. This is necessary because if the element existed in the same namespace as other elements, the content model would be non-deterministic. The reason is because a validation program would not be able to determine if the choice alternative element is a defined element or an extension element.

Open Type

An *Open Type* as defined in the X.680 standard is specified as a reference to a *Type Field* in an *Information Object Class*. The most common form of this is when the *Type* field in the built-in TYPE-IDENTIFIER class is referenced as follows:

```
TYPE-IDENTIFIER.&Type
```

See the section in this document on Information Objects for a more detailed explanation.

A reference to an *Open Type* within a SEQUENCE or CHOICE construct is converted into an XSD *any* element type. Note that the conversion is only done if the element is in one of these constructs. An open type declaration on its own has no equivalent XSD type and is therefore ignored.

An example showing how an open type might be referenced in a SEQUENCE type and the corresponding conversion to XSD is as follows:

```
SeqWithOpenType ::= SEQUENCE {
   anOpenTypeTYPE-IDENTIFIER.&Type
}
```

Generated XSD type:

```
<xsd:complexType name="SeqWithOpenType">
  <xsd:sequence>
    <xsd:any/>
  </xsd:sequence>
</xsd:complexType>
```

In this case, any valid XML instance can be used in the type. Note that the ASN.1 element name (*anOpenType*) is ignored.

Tagged Type

In ASN.1, it is possible to create new custom types using ASN.1 tag values as identifiers. These identifiers are built into BER or DER encoded messages. In general, these tags have no meaning in an XSD representation of an ASN.1 type that is used to create or validate XML markup. However, if the schema definition is to be used to generate a BER or DER instance of a type, the tag information will be required. For this reason, it is possible to add an application information annotation (*appinfo*) to the generated XSD type.

The annotation carries all of the information an application would need to know to encode a BER or DER message of the given type. This includes the tag's class, identifier number, and how it is applied (IMPLICIT or EXPLICIT). The type that specifies this information is the *TagInfo* type in the Objective Systems' XSD class library.

The mapping of an ASN.1 tagged type to XSD is as follows:

ASN.1 production:

```
TypeName ::= Tagging [ TagClass TagID ] ASN1Type
```

Generated XSD code:

```
<xsd:complexType name="TypeName" asn1:tag="[TagClass TagID]"
   asn1:tagging="EXPLICIT">
   equivalent XSD type mapping for ASN1Type
</xsd:complexType>
```

Previous versions of ASN1C generate an annotation for tag information. This can still be generated with the current version by using the *-appinfo* option.

Generated XSD code with annotation:

```
<xsd:complexType name="TypeName">
   <xsd:annotation/>
      <xsd:appinfo>
         <asn1:TagInfo>
            <asn1:TagClass> TagClass </asn1:TagClass>
            <asn1:TagID> TagID </asn1:TagID>
            <asn1:Tagging> Tagging </asn1:Tagging>
         </asn1:TagInfo>
      </xsd:appinfo>
```

```
    </xsd:annotation>
    equivalent XSD type mapping for ASN1Type
</xsd:complexType>
```

Tagging in the definition above is optional. If present, it is equal to either the keyword EXPLICIT or IMPLICIT. The default value is EXPLICIT. A default value for all types in a module can also be specified in the ASN.1 module header.

The tag's form (constructed or primitive) is not specified in the mapping above. This is because this can be determined by an application that is encoding or decoding a message of the given type.

EXTERNAL and EmbeddedPDV Type

The EXTERNAL and EmbeddedPDV types are built-in ASN.1 types that make it possible to transfer a value of a different encoding type within an ASN.1 message. These are constructed types built into the ASN.1 standard. An XSD representation of each of these types is available in the *osxsdlib.xsd* library. The ASN1C compiler generates a reference to the types in the library when it encounters a reference to one of these types.

ASN.1 production:

```
TypeName ::= EXTERNAL
```

Generated XSD code:

```
<xsd:complexType name="TypeName" base="asn1:EXTERNAL"/>
```

ASN.1 production:

```
TypeName ::= EMBEDDED PDV
```

Generated XSD code:

```
<xsd:complexType name="TypeName" base="asn1:EmbeddedPDV"/>
```

ASN1C90

The ASN1C90 version of the compiler is a separate executable that contains extensions to handle the older 1990 version of ASN.1. Although this version is no longer supported by the ITU-T, it is still in use today. This version of the compiler also contains logic to parse some common MACRO definitions that are still in widespread use despite the fact that MACRO syntax was retired with this version of the standard. The types of MACRO definitions that are supported are ROSE OPERATION and ERROR and SNMP OBJECT-TYPE.

ROSE OPERATION and ERROR

ROSE stands for "Remote Operations Service Element" and defines a request/response transaction protocol in which requests to a conforming entity must be answered with the result or errors defined in operation definitions. Variations of this are used in a number of protocols in use today including CSTA and TCAP.

The definition of the ROSE OPERATION MACRO that is built into the ASN1C90 version of the compiler is as follows:

```
OPERATION MACRO ::=
BEGIN
  TYPE NOTATION           ::= Parameter Result Errors LinkedOperations
  VALUE NOTATION          ::= value (VALUE INTEGER)
  Parameter               ::= ArgKeyword NamedType | empty
  ArgKeyword              ::= "ARGUMENT" | "PARAMETER"
  Result                  ::= "RESULT" ResultType | empty
  Errors                  ::= "ERRORS" "{"ErrorNames"}" | empty
  LinkedOperations        ::= "LINKED" "{"LinkedOperationNames"}" | empty
  ResultType              ::= NamedType | empty
  ErrorNames              ::= ErrorList | empty
  ErrorList               ::= Error | ErrorList "," Error
  Error                   ::= value(ERROR)        -- shall reference an error value
                              | type              -- shall reference an error type
                                                  -- if no error value is specified
  LinkedOperationNames    ::= OperationList | empty
  OperationList           ::= Operation | OperationList "," Operation
  Operation               ::= value(OPERATION)    -- shall reference an operation value
                              | type              -- shall reference an operation type
                                                  -- if no operation value is specified
  NamedType               ::= identifier type | type

END
```

This MACRO does not need to be defined in the ASN.1 specification to be parsed. In fact, any attempt to redefine this MACRO will be ignored. Its definition is hard-coded into the compiler.

What the compiler does with this definition is uses it to parse types and values out of OPERATION definitions. An example of an OPERATION definition is as follows:

```
login OPERATION
ARGUMENT SEQUENCE { username IA5String, password IA5String }
RESULT   SEQUENCE { ticket OCTET STRING, welcomeMessage IA5String }
ERRORS { authenticationFailure, insufficientResources }
::= 1
```

In this case, there are two embedded types (an ARGUMENT type and a RESULT type) and an integer value (1) that identifies the OPERATION. There are also error definitions.

The ASN1C90 compiler generates two types of items for the OPERATION:

1. It extracts the type definitions from within the OPERATION definitions and generates equivalent C/C++ structures and encoders/decoders, and

2. It generates value constants for the value associated with the OPERATION (i.e., the value to the right of the '::=' in the definition).

The compiler does not generate any structures or code related to the OPERATION itself (for example, code to encode the body and header in a single step). The reason is because of the multi-layered nature of the protocol. It is assumed that the user of such a protocol would be most interested in doing the processing in multiple stages, hence no single function or structure is generated.

Therefore, to encode the login example the user would do the following:

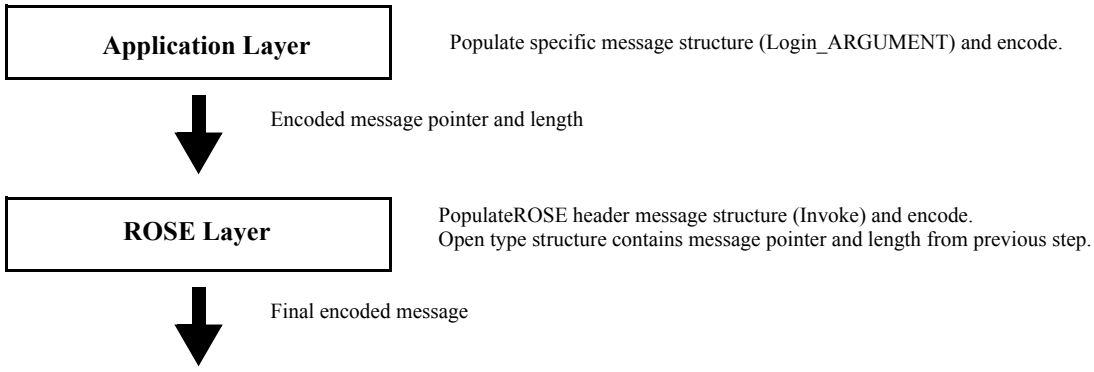
1. At the application layer, the Login_ARGUMENT structure would be populated with the username and password to be encoded.

2. The encode function for Login_ARGUMENT would be called and the resulting message pointer and length would be passed down to the next layer (the ROSE layer).

3. At the ROSE layer, the Invoke structure would be populated with the OPERATION value, invoke identifier, and other header parameters. The parameter.numocts value would be populated with the length of the message passed in from step 2. The parameter.data field would be populated with the message pointer passed in from step 2.

4. The encode function for Invoke would be called resulting in a fully encoded ROSE Invoke message ready for transfer across the communications link.

The following is a picture showing this process:


Application Layer
Populate specific message structure (Login_ARGUMENT) and encode.
Encoded message pointer and length
ROSE Layer
PopulateROSE header message structure (Invoke) and encode.
Open type structure contains message pointer and length from previous step.
Final encoded message

On the decode side, the process would be reversed with the message flowing up the stack:

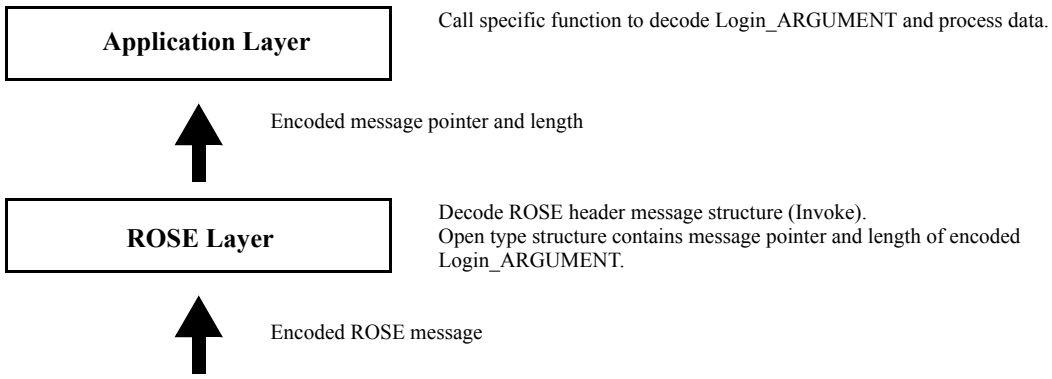
1. At the ROSE layer, the header would be decoded producing information on the OPERATION type (based on the MACRO definition) and message type (Invoke, Result, etc..). The invoke identifier would also be available for use in session management. In our example, we would know at this point that we got a login invoke request.

2. Based on the information from step 1, the ROSE layer would know that the Open Type field contains a pointer and length to an encoded Login_ARGUMENT component. It would then route this information to the appropriate processor within the Application Layer for handling this type of message.

3. The Application Layer would call the specific decoder associated with the Login_ARGUMENT. It would then have available to it the username/password the user is logging in with. It could then do whatever application-specific processing is required with this information (database lookup, etc.).

4. Finally, the Application Layer would begin the encoding process again in order to send back a Result or Error message to the Login Request.

A picture showing this is as follows:


Application Layer
Call specific function to decode Login_ARGUMENT and process data.
Encoded message pointer and length
ROSE Layer
Decode ROSE header message structure (Invoke).
Open type structure contains message pointer and length of encoded Login_ARGUMENT.
Encoded ROSE message

The login OPERATION also contains references to ERROR definitions. These are defined using a separate MACRO that is built into the compiler. The definition of this MACRO is as follows:

```
ERROR MACRO ::=
BEGIN
  TYPE NOTATION     ::= Parameter

  VALUE NOTATION    ::= value (VALUE INTEGER)

  Parameter         ::= "PARAMETER" NamedType | empty

  NamedType         ::= identifier type | type

END
```

In this definition, an error is assigned an identifying number as well as on optional parameter type to hold parameters associated with the error. An example of a reference to this MACRO for the `authenticationFailure` error in the login operation defined earlier would be as follows:

```
applicationError ERROR
PARAMETER SEQUENCE {
  errorText IA5String
}     }
::= 1
```

The ASN1C90 compiler will generate a type definition for the error parameter and a value constant for the error value. The format of the name of the type generated will be "<name>_PARAMETER" where <name> is the ERROR name (applicationError in this case) with the first letter set to uppercase. The name of the value will simply be the ERROR name.

SNMP OBJECT-TYPE

The SNMP OBJECT-TYPE MACRO is one of several MACROs used in Management Information Base (MIB) definitions. It is the only MACRO of interest to ASN1C because it is the one that specifies the object identifiers and data that are contained in the MIB.

The version of the MACRO currently supported by this version of ASN1C can be found in the SMI Version 2 RFC (RFC 2578). The compiler generates code for two of the items specified in this MACRO definition:

1. The ASN.1 type that is specified using the SYNTAX command, and

2. The assigned OBJECT IDENTIFIER value

For an example of the generated code, we can look at the following definition from the UDP MIB:

```
udpInDatagrams OBJECT-TYPE
    SYNTAX      Counter32
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
            "The total number of UDP datagrams delivered to UDP users."
    ::= { udp 1 }
```

In this case, a type definition is generated for the SYNTAX element and an Object Identifier value is generated for the entire item. The name used for the type definition is "<name>_SYNTAX" where <name> would be replaced with the OBJECT-TYPE name (i.e., udpInDatagrams). The name used for the Object Identifier value constant is the OBJECT-TYPE name. So for the above definitions, the following two C items would be generated:

```
typedef Counter32 udpInDatagrams_SYNTAX;

ASN1OBJID udpInDatagrams = {
   8,
   { 1, 3, 6, 1, 2, 1, 7, 1 }
} ;
```

Index

configuration file element 177
configuration variable 81
configuration variable 114, 134

Symbols

%ASN prefix 29

Numerics

16-bit character string 60
32-bit character string 60
8-bit character string
    derivation 59

A

ABSTRACT-SYNTAX
    C/C++ mapping 74
accessing encoded message components 103
addEventHandler method 188
ANSI-standard source code, for base run-time libraries 22
ANY or ANY DEFINED BY constructs 58
ASN.1 run-time library 21
ASN1C GUI
    ASN.1 file selection 12
    Output directory name 17
ASN1C90 213
    ROSE OPERATION and ERROR 213
    SNMP OBJECT TYPE 215
ASN1CTXT argument 114, 115, 134, 135
ASN1Object structure 59
ASN1TPDU base class 179
ASN1XERSAXHandler base class 81
attribute
    global level 24
    module level 26
    production level 27, 28
    specified in more than one section 23

B

Basic Encoding Rules 1, 5
ber command line option 5
BER decode function
    decoding a series of messages using C++ control class interface 120
    generated C function format and calling parameters 114
    generated C++ decode method format and calling parameters 114
    generated stream-oriented C function format and calling parameters 134
    performance consideration of dynamic memory management 123
    procedure for calling C decode functions 115
    procedure for calling in C 115
    procedure for using C++ control class decode method 119
BER decode function See also BER/DER C decode functions
BER encode function
    encoding a series of messages using C++ control class interface 109, 112
    encoding a series of messages using stream-oriented C++ control class interface 132
    encoding a series of messages using the stream-oriented C encode functions 129
    generated C function format and calling parameters 105
    generated C++ encode method format and calling parameters 106
    procedure for calling C encode functions 106
    procedure for calling in C 106
    procedure for using C++ control class encode method 110
    procedure for using C++ control class stream-oriented encode method 131
BER encode function See also BER/DER C encode functions
BER encoded message, diagram 106
BER stream-oriented decode function
    decoding a series of message 117, 137
    decoding a series of messages using C++ control class interface 142
    decoding a series of messages using the stream-oriented C decode functions 117, 137
    generated C++ decode method format and calling parameters 139
    procedure for calling C stream-oriented decode functions 135
    procedure for calling in C 135
    procedure for using C++ control class decode method 140
BER stream-oriented decode function See also BER/DER C decode functions
BER stream-oriented encode function
    generated C++ encode method format and calling parameters 131
    generated stream-oriented C function format and calling parameters 127
    procedure for calling C stream-oriented encode functions 128
    procedure for calling in C 128
big integers 36
binary string value specification 65
bit string
    for specifying named constants for bit positions 39
BIT STRING type
    C/C++ dynamic 37
    C/C++ mapping 37
    C/C++ static (sized) 38
    Named Bits 39

XML schema mapping 195
bitMacros command line option 5
BMPString type
C/C++ mapping 60
BOOLEAN type
C/C++ mapping 35
XML schema mapping 193

C

c command line option 5, 6
c++ command line option 5
C++ control class 80
C++ control class decode method
procedure for using in generated XER decode functions 168
C++ control class decode method, procedure for using 119
C++ control class encode method
procedure for using in generated BER encode functions 110
procedure for using in generated BER stream-oriented encode functions 131
procedure for using in generated PER encode functions 147, 154
procedure for using in generated XER encode functions 161
C++ control class interface
decoding a series of message 120, 142
decoding a series of PER messages 156
encoding a series of BER messages 109, 112, 132
encoding a series of PER messages 151
C++ control class stream-oriented decode method, procedure for using 140
C/C++ mapping
ABSTRACT-SYNTAX 74
BIT STRING 37
BMPString 60
BOOLEAN 35
Character strings 59
CHOICE 55
ENUMERATED 42
EXTERNAL 61
GeneralizedTime 60
INTEGER 35
NULL 43
OBJECT IDENTIFIER 44
OCTET STRING 40
Open Type 58
REAL 45
RELATIVE-OID 45
SEQUENCE 45
SEQUENCE OF 52
SET 51
SET OF 55
TYPE-IDENTIFIER 74
UniversalString 60
UTCTime 60
calling C BER or DER decode functions 115
calling C BER or DER encode functions 106
calling C BER stream-oriented decode functions 135
calling C BER stream-oriented encode functions 128
calling C PER decode functions 152
calling C PER encode functions 145
calling C XER decode functions 166
calling C XER encode functions 159, 165
calling C++ XER decode functions 168
calling Java PER encode methods 87, 91
calling parameters
generated C for BER decode function 114
generated C for BER encode function 105
generated C for BER stream-oriented decode function 134
generated C for BER stream-oriented encode function 127
generated C for PER decode function 151
generated C for PER encode function 144
generated C for XER decode function 165
generated C for XER encode function 158
generated C++ for BER decode function 114
generated C++ for BER encode function 106
generated C++ for BER stream-oriented decode function 139
generated C++ for BER stream-oriented encode function 131
generated C++ for PER decode function 152
generated C++ for PER encode function 144
generated C++ for XER encode function 159
case, importance in syntax errors 29
Character string type
C/C++ mapping 59
XML schema mapping 198
character string value specification 65
CHOICE type
C/C++ mapping 55
populating generated choice structures 57
XML schema mapping 209
CLASS
generated C code 69
generated C++ code 70
New generated assignments 72
OPTIONAL keyword 72
command line options 5
commas, when to use 29
compact command line option 6
compacting code 6
compare functions
function name format 176
generated 176
source file for 6
compat command line option 6
compiler
error reporting 29
running 3

compiling generated code 21
config command line option 6
configuration file 23
configuration specifications
    examples 24
configuration specifications See also attribute
constants, for named bits 39
contents method 185
copy functions
    function name format 177
    generated 177

D

decode function
    prototype 79
Decode method 80
decode method
    C++ control class 119
    in generated C/C++ source code 82
decode method, C++ control class
    using in generated XER decode functions 168
DecodeFrom method 80
DEFAULT keyword in SEQUENCE 49, 204
deferred decoding 125
der command line option 6
DER decode function
    procedure for calling in C 115
DER decode function See also BER/DER C decode functions
DER encode function
    procedure for calling in C 106
DER encode function See also BER/DER C encode functions
diagnostic messages, adding to generated code 10
directory
    generated files 9
    searching for IMPORT items 8
directory tree, for porting run-time code 22
Distinguished Encoding Rules 1, 6
dynamic encode buffer 106, 108
    for BER encoding 111
    for PER encoding 147, 150
dynamic memory management 98
    performance considerations in generated BER decode functions 123
    performance considerations in generated PER decode functions 157
Dynamic SEQUENCE OF type 52

E

Embedded PDV type
    XML schema mapping 212
encode buffer, dynamic
    for BER encoding 111
    for PER encoding 147, 150
encode buffer, static
    for BER encoding 106, 108, 111
    for PER encoding 150
encode function
    BER 105
    prototype 79
Encode method 80
encode method, C++ control class
    using in generated BER encode functions 110
    using in generated BER stream-oriented encode functions 131
    using in generated PER encode functions 147, 154
    using in generated XER encode functions 161
encode method, using in generated C/C++ source code 82
encode/decode functions
    source file for 6
    suppressing 8
encoded message
    accessing 103
EncodeTo method 80
encoding data 98
endElement event 185
ENUMERATED type
    C/C++ mapping 42
    XML schema mapping 200
enumPrefix attribute 28
error
    semantic 29
    syntax 29
error event 185
error macro, ROSE 215
error reporting, compiler 29
event
    endElement 185
    error 185
    startElement 185
event handler interface
    example-error handler class 188, 190
    example-formatted print handler 186
    how it works 185
    how to use it 186
events command line option 6
EXPAT interface library 169
Expat XML Parser 163
export of types 192
extension elements in CHOICE 210
extension elements in SEQUENCE 50, 204
EXTERNAL type
    C/C++ mapping 61
    XML schema mapping 212

F

fast copy 124
field
    fixed type 83
    variable type 84
file, platform.mk 22

fixed type field 83
freeing allocated memory 115, 118, 120, 122, 139, 141, 143
function, encode/decode prototypes 79

G

-genCopy 51
GeneralizedTime type
  C/C++ mapping 60
  XML schema mapping 199
generated BER decode function
  decoding a series of messages using C++ control class interface 120
  generated C function format and calling parameters 114
  generated C++ decode method format and calling parameters 114
  performance consideration of dynamic memory management 123
  procedure for calling C decode functions 115
  procedure for using C++ control class decode method 119
generated BER encode function
  encoding a series of messages using C++ control class interface 109, 112
  generated C function format and calling parameters 105
  generated C++ encode method format and calling parameters 106
  procedure for calling C encode functions 106
  procedure for using C++ control class encode method 110
generated BER stream-oriented decode function
  decoding a series of messages using C++ control class interface 142
  decoding a series of messages using the stream-oriented C decode functions 117, 137
  generated C function format and calling parameters 134
  generated stream-oriented C++ decode method format and calling parameters 139
generated BER stream-oriented encode function
  encoding a series of messages using C++ control class interface 132
  encoding a series of messages using the stream-oriented C encode functions 129
  generated stream-oriented C function format and calling parameters 127
  generated stream-oriented C++ encode method format and calling parameters 131
  procedure for calling C stream-oriented encode functions 128
  procedure for using stream-oriented C++ control class encode method 131
generated C code for Table Constraints 85
generated C function format
  BER decode method 114
  BER encode method 105
  BER stream-oriented decode method 134
  PER decode method 151
  PER encode method 144
  XER decode method 165
  XER encode method 158
generated C table constraint function prototypes 85
generated C++ class definition for BER/DER/PER 80
generated C++ class definition for XER 81
generated C++ class for ASN.1 information object set 78
generated C++ class for ASN.1 information objects 75
generated C++ classes for table constraints 86
generated C++ copy constructor 179
generated C++ decode method format
  BER decode method 114
  PER decode method 152
generated C++ destructor 179
generated C++ encode method format
  BER encode method 106
  PER encode method 144
  XER encode method 159
generated C++ stream-oriented decode method format
  BER decode method 139
generated C++ stream-oriented encode method format
  BER encode method 131
generated C++ table constraint function prototypes 86
generated C/C++ source code
  ASN1C90 213
  BER stream-oriented decode functions 134
  BER stream-oriented encode functions 127
  event handler interface 185
  generated BER decode functions 114
  generated BER encode functions 105
  generated compare methods 176
  generated copy methods 177
  generated PER decode functions 151
  generated PER encode functions 144
  generated print methods 173, 181, 183
  generated XER decode functions 163
  generated XER encode functions 158, 170
  header file 31
  IMPORT/EXPORT of types 192
  ROSE OPERATION and ERROR 213
  SNMP OBJECT TYPE 215
generated compare functions 176
generated copy functions 177
generated information object definition 82
generated methods 82
generated PER decode function
  generated C function format and calling parameters 151
  generated C++ decode method format and calling parameters 152
  performance consideration of dynamic memory management 157

procedure for calling C decode functions 152
generated PER encode function
decoding a series of messages using C++ control class interface 156
encoding a series of messages using C++ control class interface 151
generated C function format and calling parameters 144
generated C++ encode method format and calling parameters 144
populating generated structure variables for encoding 145
procedure for calling C encode functions 145
procedure for using C++ control class encode method 147, 154
generated print functions 173, 181, 183
generated stream-oriented BER decode function
procedure for calling C decode functions 135
procedure for using C++ control class decode method 140
generated stream-oriented C function
encoding a series of BER messages 129
generated stream-oriented C function format
BER stream-oriented encode method 127
generated stream-oriented C++ encode method format
BER encode method 131
generated structure variables
populating for encoding 102
populating for PER encoding 145
populating for XER encoding 159
generated XER decode function
generated C function format and calling parameters 165
procedure for calling C decode functions 166
procedure for calling C++ decode functions 168
procedure for using C++ control class encode method 168
generated XER encode function
generated C function format and calling parameters 158
generated C++ encode method format and calling parameters 159
populating generated structure variables for encoding 159
procedure for calling C encode functions 159, 165
procedure for using C++ control class encode method 161
generated XML schema source code 193
getCopy method 177
GetMsgCopy method 111
getMsgCopy method 111
GetMsgPtr method 110
global level attributes 24
graphical user interface 11
GUI Wizard 11

H

h command line option 8
header file 8
differences between C and C++ versions 33
sample from a C header file 31
sample from a C++ header file 32
hexadecimal string value specification 65
hyphens See special characters, invalid

I

I command line option 8
import of types 192
include imports attribute 26
Information Object
generated C code 75
generated C++ code 75
generated type assignments 76
information object definition, generated 82
Information Object Set
generated C code 77
generated C++ code 78
information objects type definition 82
integer
for holding bit number 37
size, big integer 36
INTEGER type
C/C++ mapping 35
XML schema mapping 194
INTEGER type, large integer support 36
INTEGER value specification 64
isBigInteger attribute 28
isPDU attribute 28
ITU X.680 ASN.1 sstandard 5

L

lax command line option 8
library
run time 21
libxml2 interface library 169
linking generated code 21
list command line option 8
list-based SEQUENCE OF type, generating 53
lowercase letters, when to use 29

M

macro
ROSE OPERATION 5, 83
ROSE OPERATION and ERROR 213
SNMP OBJECT TYPE) 215
memory
dynamic 98
memory management
allocating variables on the stack 102
using C malloc and free functions 103
using run-time library functions 103

memory management, dynamic
performance considerations in generated BER decode functions 123
performance considerations in generated PER decode functions 157
messages
BER encoded, diagram 106
repetitive BER encoding 109, 112, 129
repetitive PER encoding 151
method
contents 185
generated 82
GetMsgCopy 111
getMsgPtr 110
minInclusive/maxInclusive facets 195
minLength/maxLength facets 195, 198
minOccurs/maxOccurs facets 203, 207
module level attributes 26
module, specification 23
MSXML interface library 169

N

named bit constants 40
newCopy method 178
nibble-allocation algorithm 98
nodecode command line option 8
noencode command line option 8
noIndefLen command line option 9
noOpenExt command line option 9
noPDU attribute 27
NULL type
C/C++ mapping 43
XML schema mapping 201

O

o command line option 9
OBJECT IDENTIFIER type
C/C++ mapping 44
XML schema mapping 201
object identifier value specification 66
OCTET STRING type
C/C++ dynamic 41
C/C++ mapping 40
C/C++ static (sized) 41
XML schema mapping 197
octet, for holding bit string contents 37
Open type
C/C++ mapping 58
XML schema mapping 210
OPTIONAL keyword in SEQUENCE 48, 203
osxsdlib.xsd 195

P

Packed Encoding Rules 1, 9
parameterized type definition 62
parse errors, finding by generating a listing 8
parsing process
diagram of significant events 185
events passed to user 185
pattern constraint 199
per command line option 9
PER encode function
format of generated prototype 79
procedure for calling in C 145
PER encode method
procedure for calling 87, 91
permitted alphabet constraint 199
platform.mk, editing 22
populating generated structure variables
for PER encoding 145
for XER encoding 159
populating generated structure variables for encoding 102
Porting Run-time Code to Other Platforms 22
prefix
%ASN 29
ASN1C_ 33
ASN1T_ 35, 56
ASN1V_ 64, 65
enumPrefix 28, 42
for generated BER decode function 114, 134
for generated BER encode function 105, 127
for generated C/C++ source code 173
for generated PER decode function 151
for generated PER encode function 144
for PER generated prototypes 79
for PER prototypes 32
type (for attirubtes specified in more than one section 23
valuePrefix 27
print functions
function name format 173, 181
generated 173, 181, 183
source file for 7
print-to-string functions
source file for 7, 8
procedures for encoding data 98
production level attributes 27, 28
production, specification 23
Protocol Data Unit (PDU) 81

R

REAL type
C/C++ mapping 45
XML schema mapping 202
RELATIVE-OID type
C/C++ mapping 45
XML schema mapping 202
Remote Operations Service Element (ROSE) 213
ROSE
decode process 214
encode process 214

ERROR MACRO 215
ROSE OPERATION and ERROR 213
ROSE OPERATION macro 5, 83
rtInitContext function 105, 106, 166
rtSetFastCopy function 125
rtStreamInit function 165, 166
rules
Basic Encoding Rules 5
Distinguished Encoding Rules 6
Packed Encoding Rules 9
XML Encoding Rules 10
run-time code, porting to other platforms 22
run-time library, ASN.1 21

S

SAX API 169
SAX content handler interface 165
semantic errors 29
SEQUENCE
value specification 67
SEQUENCE OF type
C/C++ mapping 52
dynamic 52
generating temporary types 53
list-based SEQUENCE OF type 53
other constructed types 54
static (sized) 53
XML schema mapping 206
SEQUENCE type
C/C++ mapping 45
DEFAULT keyword 49
extension elements 50, 204
OPTIONAL keyword 48
unnamed elements 48
XML schema mapping 203
SET OF type
C/C++ mapping 55
XML schema mapping 206
SET type
C/C++ mapping 51
XML schema mapping 206
SNMP OBJECT TYPE macro 215
source code, ANSI standard 22
source file
for encode/decode functions 6
for generated compare functions 6
for generated print functions 7
for generated print-to-string functions 7, 8
special characters, invalid 29
specification
attribute in more thanone section 23
module 23
production 23
standard, ITU X 680 5
startElement event 185
static (sized) SEQUENCE OF type definition 53
static command line option 9
static encode buffer
for BER encoding 106, 108, 111
for PER encoding 150
storage attribute 25
stream command line option 9
stream open functions 166
stream-oriented decode method
C++ control class 140
syntax errors 29

T

Table Constraint Code Generation 84
Tagged type
XML schema mapping 211
temporary types, generating for SEQUENCE OF type definition 53
toString method 174
trace command line option 10
tree, directory 22
TYPE-IDENTIFIER 58, 210
C/C++ mapping 74
types, import and export 192

U

UniversalString type
C/C++ mapping 60
unnamed elements in SEQUENCE 48
uppercase letters, when to use 29
UTCTime type
C/C++ mapping 60
XML schema mapping 199
UTF-8 string data 60
UTF8String type
C/C++ mapping 59

V

value specification
binary string 65
character string 65
CHOICE 68
hexadecimal string 65
INTEGER 64
object identifier 66
SEQUENCE 67
SEQUENCE OF 67
SET 67
SET OF 68
type definition 64
valuePrefix attribute 27
variable type field 84
version 5.x compatible code, generating 6
VisibleString type
C/C++ mapping 59

W

warnings command line option 10
warnings, output information 10

X

xer command line option 10
XER decode function
    procedure for calling in C++ 168
    procedure for using other C++ XML Parsers 169
XER encode function
    procedure for calling in C 159, 165
    procedure for calling in C++ 161
XERCES interface library 169
XML Encoding Rules 1, 10
XML Parser 163
XML SAX content handler interface 82
XML schema mapping
    Big Integer 194
    BIT STRING 195
    BOOLEAN 193
    Character String 198
    CHOICE 209
    Embedded PDV 212
    ENUMERATED 200
    EXTERNAL 212
    GeneralizedTime 199
    IMPORT statements 193
    INTEGER 194
    MODULE definition 193
    NULL 201
    OBJECT IDENTIFIER 201
    OCTET STRING 197
    Open Type 210
    REAL 202
    RELATIVE-OID 202
    SEQUENCE 203
    SEQUENCE OF 206
    SET 206
    SET OF 206
    Tagged Type 211
    UTCTime 199
xsd command line option 10